unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: 87-24-04	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): Northwest Laboratory for Integrated Systems (formerly UW/NW VLSI Consortium) Semiannual Technical Report No. 4	5. TYPE OF REPORT & PERIOD COVERED: Technical, interim
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Northwest Laboratory for Integrated Systems	8. CONTRACT OR GRANT NUMBER(s): MDA903-85-K-0072 ARPA-4563/2 Code 5D30
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Northwest Laboratory for Integrated Systems University of Washington, Computer Science Seattle, WA 98195 FR-35	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: DARPA - ISTO 1400 Wilson Boulevard Arlington, VA 22209	12. REPORT DATE: December, 1986
	13. NUMBER OF PAGES: 60
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office): ONR University of Washington 315 University District Building 1107 NE 45th St., JD-16, Seattle, WA 98195	15. SECURITY CLASS. (of this report): unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this report is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

VLSI Design Generators, MN, hercules, NC, Quarter Horse, CAM, ROM, RAM, nMOS, CMOS, CFL, Magic, Caesar, Spice

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This document reports on the research activities of the Northwest Laboratory for Integrated Systems, formerly the UW/NW VLSI Consortium, for the period of December 10, 1986 to April 6, 1987 under sponsorship of the Defense Advanced Research Projects Agency, under contract number MDA903-85-K-0072, program code number 5D30.

keywords include

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-LF-014-6601

unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

NORTHWEST LABORATORY FOR INTEGRATED SYSTEMS

Semiannual Technical Report No. 4

University of Washington

April 6, 1987
TR# 87-24-04

Reporting Period: 10 December 1986 to 6 April 1987

Principal Investigator: Lawrence Snyder

Sponsored by
Defense Advanced Research Projects Agency(DoD)
ARPA Order No. 4563/2
Issued by Defense Supply Service-Washington
Under Contract #MDA903-85-K-0072
(Program Code Number: 5D30)

Contents

1 Executive Summary

This document reports on the research activities of the Northwest Laboratory for Integrated Systems.(formerly the UW/NW VLSI Consortium) for the period 10 December 1986 to 6 April 1987 under the sponsorship of the Defense Advanced Research Projects Agency. The applicable contract for this period is MDA 903-85-K-0072.

At the beginning of calendar year 1987 we formally became the Northwest Laboratory for Integrated Systems. As our new name implies, we see our research building upon the design generator research and extending naturally to the larger issues of systems design and integration.

The design generators project reached a new milestone with the completion of both layout and schematic compilers for the declarative notation MN (see Section 2). Our experiments have shown that the notation is sufficiently flexible to accomodate a variety of circuit designs. Implementation of both a decoder and multiplier gives a compact notation and emphasizes the structural correlation between the layout and the schematic (see Appendices B and C).

Although our new name implies a new orientation, we have dealt for some time with system issues in several ongoing projects. The Quarter Horse, a 32 bit microprocessor designed by a half dozen students, presented problems not only in complexity-area tradeoffs of chip design, but also in the board-level design of a Multibus interface. The Pyramid machine, a hierarchical graphics processor, began with an nMOS chip design and is now a 64 x 64 pixel working prototype. Most recently, the winter quarter Advanced VLSI class initiated the design of a special processor for the real-time generator of parametric curves and surfaces (see Section 3).

The simulation system *nc* (*network c*) received considerable use during the recent quarter. Students in the Advanced VLSI design class employed *nc* in the functional simulation of both the chip and the system interface. The designer of the memory elements in Section 7 used *nc* as well as *spice* to verify the functionality of several analog elements. Ongoing work focuses on language enhancements and run-time diagnostics (see Section 6).

Several new high performance circuit designs have been initiated recently in support of the Advanced VLSI class. We have invested considerable effort in developing flexible generators for decoders, precharge circuitry, and sense amplifiers – common structures in a variety of memory elements. We are currently at work writing generators for a RAM (single/dual ported, column/row decoded), ROM, and a CAM.

Work continues on the power analyzer *hercules*. Designed for the analysis of both nMOS and CMOS circuits, this tool is intended to estimate load and direct currents

as well as flag excessive voltage drops between pins and devices. Initial experiments indicate current estimates within 25% of *spice* (Appendix D).

We recently completed version 3.1 of our VLSI design software, a package that includes tools from Berkeley, MIT and CMU as well as our own tools. The response has been enthusiastic – so far 75 requests have been received.

2 Progress on Design Generators

(W. Winder, R. Nottrott)

2.1 Introduction

During the first half of 1986 a declarative model for generator construction was defined. The intent of this model, which we term "Meei's Notation" (MN), is to provide a concise representation that captures the fundamental structural and functional properties of the circuit. During the past half year, we have developed back end processes that produce a variety of circuit descriptions from MN.

The back end processes are compilers which convert the declarative description MN and the accompanying leaf cells into the appropriate representation. The schematic back end *schgen* converts an MN and specially prepared post-script-like files into a post-script file suitable for printing. The layout back end *laygen* converts an MN and layout leaf cells (in either *magic* or *caesar* format) into a layout description. When complete, the functional back end *fungen* will convert a MN and leaf cells into an *nc* (*network c*) input file, which can be compiled and run with a stimulus file to create a simulation. Currently, *schgen* and *laygen* have been implemented. Work is proceeding on *fungen.*

Each compiler runs in three major steps: parsing, table building, and analyzing.

2.2 Parsing, Table Building and Analysis

The first major step in creating the output representation is construction of the parse tree. This step is driven by the input description (MN), which must be an instance of the grammar described in Appendix A. Minor changes to the grammar previously reported have been made to increase clarity, to enhance consistency with the C programming language, and to add features.

The parse tree is made of nodes, which are the non-terminal symbols of grammar instance, and leaves, which are the grammar tokens (integer literals, keywords, identifiers, etc.). All subsequent processing of the compiler analyzes this tree.

The function which builds the parse tree is created by running the grammar (in *yacc* form) through a *yacc* to *yacc* translator *ykty.*[1] The output of this process is then

[1]The *yacc* to *yacc* translator *ykty* was originally developed by W. Beckett as an aid to the development of *nc*.

input to *yacc* to produce the parser. *Ykty* adds actions to the bare grammar which cause the building of the appropriate nodes and the leaves whenever a grammar rule is identified. Automating the tree building in this manner allows easy modification of the grammar. Unfortunately, since parsing and analyzing are decoupled, much care must be taken when adding the corresponding analysis functions for modified grammars.

This parse/analyze method was taken rather than the standard *yacc* integrated method (all actions within the grammar) because the tree information must be retained for the declarative description. It is quite possible to analyze sub-trees many times with different inputs, and the full sub-tree must be available for each analysis.

The global symbol table, leaf cell table, function table, and composite table (to assist evaluation of objects) are each built. The parse tree is also checked to make sure there are no meaningless constructs (for the given representation).

The exact steps performed during analysis vary from compiler to compiler, but each analysis function is written to achieve maximum compatibility. Thus, there is one function that evaluates all expressions regardless of representation, but separate functions to "access" individual leaf cells, each dependent on the representation.

In general, during analysis a top-down decomposition is performed for each expression. The expression is then composed and the result is returned as the value of the object. If a line of the MN reads:

$$A = B \mid C;$$

and A is to be evaluated, first B is evaluated, then C is evaluated, then the two (B and C) are vertically juxtaposed, with the resulting object passed back as the "value" of A.

2.3 Schematic and Layout Back End Programs

The schematic back end *schgen* produces a postscript formation representation. When leaf cells are evaluated, the corresponding picture is drawn on a virtual page. When the picture is complete, it is scaled to a physical page. The leaf cell interface has been extended to allow passing of string parameters for dynamic labeling inside the leaves. We are currently implementing the keywords "ROT" (rotate), "MX" (mirror in x) and "MY" (mirror in y).

The layout back end *laygen* is built on top of the *cfl* layout language. One major extension was necessary to implement the overlap operators. The position of two

cells aligned by overlap is determined by one or more line labels somewhere near the corresponding borders of two cells to be aligned. Thus if A is given by:

$$A = B --\hat{} C;$$

and A is to be evaluated, the vertical line labels near the right border of B are compared with the vertical line labels near the left border of C. Labels are paired, if possible, by the text of the label (must match exactly). The lengths of corresponding line labels must be the same. If all the conditions are met, A is formed by aligning a label in B on top of the corresponding label in C. This alignment is implemented using two new functions in *cfl*, *sx* (signal align in the x direction) and *sy* (signal align in the y direction).

2.4 Software Development

The compilers were developed with the idea of trying to reuse as much code as possible from compiler to compiler, i.e. to maximize the general functions and minimize the specific functions for each. To that end, most of the analysis code is used in both of the implemented compilers, even though some particular functions may not be appropriate (e.g. overlap in *schgen*). The source code is used unmodified, only having to be recompiled with different header files (environmental differences). Heavy use was also made of *ykty* and some utility functions which were created for the *nc* program. To date, the following gives the source lines of code (SLOC) for each functional area (to the nearest 500 SLOC):

Function	SLOC
Parser (including grammar, lexical analyzer and *ykty*)	3000
Headers and Common Analysis	5000
Utilities and Debug Facilities	3000
schgen Specific	1000
post-script support (used in schgen)	1000
laygen Specific	500
cfl (used in laygen)	16000
Total:	29500

3 A Graphics Accelerator for Curves and Surfaces

(C. Ebeling)

The Advanced VLSI Design class was taught during the winter quarter as the second of a two quarter sequence. This year we chose to implement a chip for drawing parametric curves and surfaces that will be used to support an interactive graphical design system. This work was done in collaboration with Tony DeRose who has developed a fast method for drawing generalized parametric curves based on deCasteljau's algorithm.

Parametric curves (and surfaces) can be specified by a set of control points that somehow characterize the shape of the desired curve, and a method of generating a curve using these control points. There are many methods to generate such curves, each of which interpolates the points differently. In general, however, the curve can be represented by the formula:

$$Q(t) = \sum_i V_i B_i(t)$$

where t varies over some interval $[t_i, t_f]$. In this equation, V_i are the control points and B is the blending function that generates the curve using the control points. The number of control points determines the degree of the curve, with curves of degree 3 being the most common.

One example of a parametric curve is the Bezier curve, whose blending function is given by:

$$B_i^d(t) = \binom{d}{i} t^i (1-t)^{d-i}, \; t \in [0,1]$$

This function can be computed using a triangular array of processors according to deCasteljau's algorithm. The control points are presented to the base of the triangle and are combined according to the blending function as they move through the triangle, resulting in a single point on the curve corresponding to the value of t. The entire curve can be generated by varying the value of t from 0 to 1. This method takes advantage of parallelism both in terms of the number of processors in the array and in terms of pipelining within the processors. Each processor performs what is essentially a linear interpolation between two inputs, which can be done with a single multiplication and two additions. Thus it is feasible to build an array of processors to compute degree 3 and 4 curves on a single chip.

As DeRose has shown, this method can be extended to other parametric curves by making the multipliers in each of the processing elements arbitrary linear functions.

In the Bezier case these functions are t and $1 - t$, but in general they can be $L(t)$ and $R(t)$ under the restriction that $L(t) + R(t) = 1$. Uniform B-splines, non-uniform B-splines and Lagrange curves can be generated by suitable choices for $L(t)$ and $R(t)$.

The class first studied a range of implementation issues and settled on two distinct architectures. The first uses a straightforward implementation of the triangular array with as many processing elements as possible. This allows the greatest amount of parallelism but limits the degree of the curve to that allowed by the number of processing elements one can fit on a chip. One team of students has designed a chip based on this architecture for degree 4 curves. It contains the control necessary to generate both x and y coordinates when generating curves and x, y and z coordinates when generating surfaces. The processing elements use 16x16 modified Booth multipliers and 4 pipelined stages to achieve a throughput of 5 million points per second. The information that describes a curve, ie. the control points and the linear function coefficients, is double-buffered so that the generation of the next curve can begin as the previous curve is finished.

The second team implemented a more general architecture that allows curves of greater degree. This chip uses just one high-performance processing element along with many pipeline registers and programmable control. This chip will be somewhat slower than the first chip but will be more flexible both with respect to the degree of the curve generated and the actual processing method.

The I/O interface to these chips has been designed so that internal data and control registers can be written and read directly from a processor bus. We plan to use a standard VME board to install a graphics board based on these chips directly in a Sun3/160 workstation. The system may then be used by the graphics group for further research into interactive design.

4 VLSI Tools Release 3.1

(L. McMurchie, W.Jessop)

Included in this release are all of the Berkeley '86 tools, in particular *magic*. Although we use *magic* extensively, we have included many of the older Berkeley tools which may have utility in special situations. *Caesar*, for example, is useful for creating actual mask geometries.

We have expanded the number of graphics terminals/workstations which *magic* supports. In addition to the SUN color workstation drivers distributed with the '86 Berkeley tools, there is support for the VAXstation GPX (drivers developed at Stanford) as well as the Apollo DN series. All of these implementations require eight bit planes for effective use of color. Also supported are AED and Metheus Omega color graphics devices.

We have modified our layout assembly system *cfl* (Coordinate Free LAP) so that it tangos with *magic* (i.e. it reads and writes .mag files). *Cfl* treats *magic*'s abstract layers just like actual mask layers. Modifications to the *dbx* debugger allow one to display symbols interactively in the debug run of a *cfl* program.

While we use *magic* extensively to create layouts, we continue to use *mextra* to extract CIF files created with *magic*, as opposed to *magic*'s internal extractor. In fact we have modified *mextra* so that it treats p-well and n-substrate as conducting layers. (an option also exists to treat wells and substrates as nonconducting layers) We found this necessary for detecting GND-Vdd shorts created by connecting an ohmic contact to the wrong rail. *Mextra* also produces an additional file (.tbs) which contains for every device all four terminals (gate, source, drain and substrate). The program *valtbs* examines this file to insure that all p channel devices have substrates connected to GND, and n channel devices have substrates connected to Vdd. Any devices in wells without ohmic contacts are flagged. An added feature of treating the wells as conducting layers is that the input pads, which commonly contain a well resistor may now be simulated from the actual bonding pad, instead of from inside the guard ring.

Our emphasis here at the UW is in the bulk 3 micron CMOS and scalable CMOS technologies supplied by MOSIS. *Caesar*, *lyra*, *cfl*, and *cdrc* support the 3 micron bulk CMOS process. *Magic* and *cfl* support the scalable CMOS process. *Mextra* supports both. We have derived "slow", "typical" and "fast" sets of parameters for both *spice* and *rnl* based on information from MOSIS. We have expanded the meaning of the .config file so it acts as a source of process parameters for both *spice* and *rnl*.

We have developed a number of layout generators using *cfl* and include them in this release. They vary considerably in robustness. Some are compatible with the MOSIS

3 micron bulk CMOS process; others are compatible with the scalable CMOS rules. We have found the padframe generator *pads* especially useful. From a simple input file a fully stuffed frame is generated, with input, output, or tristate pads placed according to the user's requirements. The frame is consistent with the MOSIS standard frame specifications. Other generators include a multiplier, decoder, ROM, RAM, MUX, buffer, counter, and several varieties of shift registers. Instances created by some of these generators have been fabricated; other generators are currently in the process of being validated.

At the present time we have received a total of 76 requests for Release 3.1.

5 Educational Offerings

As part of an ongoing series of seminars on special topics, the LIS conducted a seminar on fault simulation in December. The intent was to bring a local audience of interested engineers up-to-speed on the basics of fault simulation. This seminar was the culmination of a year's work in this field by one of the industrial liaisons to the Consortium, Kanu Emeruwa of Microtel Pacific Research. Mr. Emeruwa successfully set up a fault grading service for a VLSI design environment, using the public domain software FMOSSIM (Bryant and Schuster 1984).

A number of outside speakers contributed their expertise to the seminar. Prof. Randal Bryant of Carnegie-Mellon University described the model of fault simulation employed by FMOSSIM. Hector Sucar of Intel Corp. gave an analysis of chip defects and the relationship of those defects to the models employed by FMOSSIM. Mr. Sucar also described the technique of functional test grading as applied to the 80386 microprocessor.

6 Simulation With Network C

6.1 Enhancements

(W. Beckett)

Over the last few months major progress on *nc* has been limited to the conversion of the analog simulation facilities from their development environment (Fortran on a CDC Cyber 855) to this environment, namely C on our VAX 780 running Unix. This work is still incomplete, since, while analog circuits seem to be simulating correctly, convergence of the numerical method seems to require about 4/3 more iterations on the VAX. This difference appears not to be a consequence of the difference in numerical precision of the machines but rather of a subtle bug in the VAX implementation.

This quarter we made the functional and MOS capabilities of *nc* available to our staff and also to a Computer Science class. The class is using it to design a special purpose graphics chip. So far, the system seems to be working well for functional level models and, even though the documentation is incomplete, has been well received by the class.

Naturally, quite a number of bugs have been discovered and fixed as a result of this "production" test. In addition, a number of excellent suggestions have been made for enhancements to the language. These suggestions, most of which have come from Carl Ebeling, include improving the diagnostic capabilities through a number of run time consistency checks and making the language more compact through a macro like capability for generating lists of element descriptors and lists of terms. We plan to implement these suggestions as soon as time permits.

The two most severe problems to be discovered concerned the plot output and MOS simulation. The plot output problem appeared as a consequence of an attempt to make the *nc* output file compatible with one of our plot programs, *simscope*. When this modification was implemented, we lost the ability to discriminate on the plot file between two signals having the same name but residing in different parts of the circuit hierarchy. The simulation is correct but the resulting plot is incorrect. The simscope interface is being reconsidered.

The second major problem was discovered in the MOS evaluator. The MOS calculation computes the response of each MOS subnetwork to input events and stores this result in a waveform table. Each signal in the waveform table is then fit with a piece-wise linear approximation. The inflection points of the piece-wise linear approximations are then queued as output events. Prior to queuing these inflection points, a check called the continuity condition is applied and, if possible, the first seg-

ment of each piece-wise linear approximation is adjusted so that the voltage on the corresponding node is a continuous function of time. The original continuity check operated by computing the intersection of the current waveform for each node with the first segment of the approximation. If the time of intersection had not yet passed the queuing of the segment was delayed until that time. This algorithm had the effect of preserving the slope of the approximation but modified its initial value and the time at which it was to occur.

As implemented, the continuity check did not take the clipping boundary into account. The clipping boundary is typically [0.0,5.0] for MOS nodes and is used to prevent node voltages from rising or falling indefinitely. Failure to consider the clipped form of the node voltage resulted in some cases of completely unrealistic times for the intersections. The result would be moved so far into the future that it would be preempted by a later event and lost altogether.

When the clipping boundary is considered, the equations for the intersections of the two lines become complex. Rather than take this approach, a new approach to maintaining continuity is being tried. This approach simply recomputes the first segment of the piece-wise linear approximation in all cases in a manner that insures continuity. The slope produced by the piece-wise linear fit procedure is no longer used. In principle, this should make the history mechanism of *nc* slightly less effective. However, in practice, the new continuity adjustment seems to be working well.

The long awaited *nc* document is both still longer and, regrettably, still awaited.

6.2 Experiments

(L. McMurchie)

RAMs are difficult circuit elements to simulate because of the high degree of coupling between all cells connected to a single bit line. Sense amplifiers often pose difficult problems as well because of their totally analog behavior. It is not surprising that switch level simulators such as *rnl* that attempt to model charge sharing are not able to obtain even correct functional behavior of these circuits. One is inevitably forced to use *spice* to simulate small parts of the overall structure.

One of our tests of *nc* at the MOS transistor level has been the simulation of exactly such a circuit – the dual-ported RAM described in Section 7.3. Figure 1 contains the *spice* simulation of the read/write cycle of the memory cell, sense amplifier and precharge circuitry. Figure 2 contains the identical experiment performed with *nc*. The node labels are as indicated in figures 2 and 4 of Section 7.3. The write cycle (WWORD0 low) writes a logic one onto NCELL0 and corresponding zero onto CELL0. The *nc* representation of of CELL0 and NCELL0 show a reasonable linear approximation of the rise and fall curves of *spice*. During the read cycle (RWORD0 high) in the *spice* run, RBIT0 and NRBIT0 show the differential behavior that is amplified into OUT0 and NOUT0. The analog behavior of both RBIT0 and NRBIT0 is difficult for *nc* to approximate, as a comparison of figures 1 and 2 shows. *Nc* does, however, model the differential behavior of these two signals sufficiently well, that OUT0 and NOUT0 are correctly driven to the rails.

The *spice* example completed in 964 seconds on a VAX 780. The *nc* experiment required 303 seconds.

Most of our experience with *nc* has been in the functional realm. Both of the designs for the graphics accelerator described in Section 6 have been simulated extensively with *nc*. Initially a crude model of the function of the graphics accelerator was implemented. Only the functionality of the chip was considered. The entire processing element was modeled with a single C language statement. Control was implemented with nested "for" loops. Data storage resided in arrays. The overall objective was to obtain the simplest working model of the chip as a starting point for further refining.

Because *nc* transforms the description of a circuit into a C program which is then compiled, linked, and executed, the user has considerable freedom in the display of the output data from the *nc* run. He can, for example, include C routines that put data out in any format desired. By triggering these routines on changes in the appropriate signals, the data can be outputted to the user or into a file. In the case of the graphics chip, we linked in a graphics library with drivers for a TEK 4010 and wrote a simple procedure that called these routines whenever the curve coordinates

were generated. An input routine asked the user for the control points describing the curve. The resulting *nc* program then was able to display on the appropriate TEK 4010 compatible device the simulated curve. The total lines of *nc* code required for this experiment was less than two hundred (not including the graphics drivers); programmer time required was a few days.

Successive refinements of this initial *nc* model have been made that describe the function of the main physical components of the chip so that the model now resembles a hardware description far more closely. The graphics interface has been retained so that it is still possible to draw the same curves as with simplest model, albeit much more slowly due to the much greater level of detail in the model.

Our experiences have indicated the need for more diagnostics to pinpoint problems with models. Also useful would be a macro facility for generating entire sets of elements and descriptors.

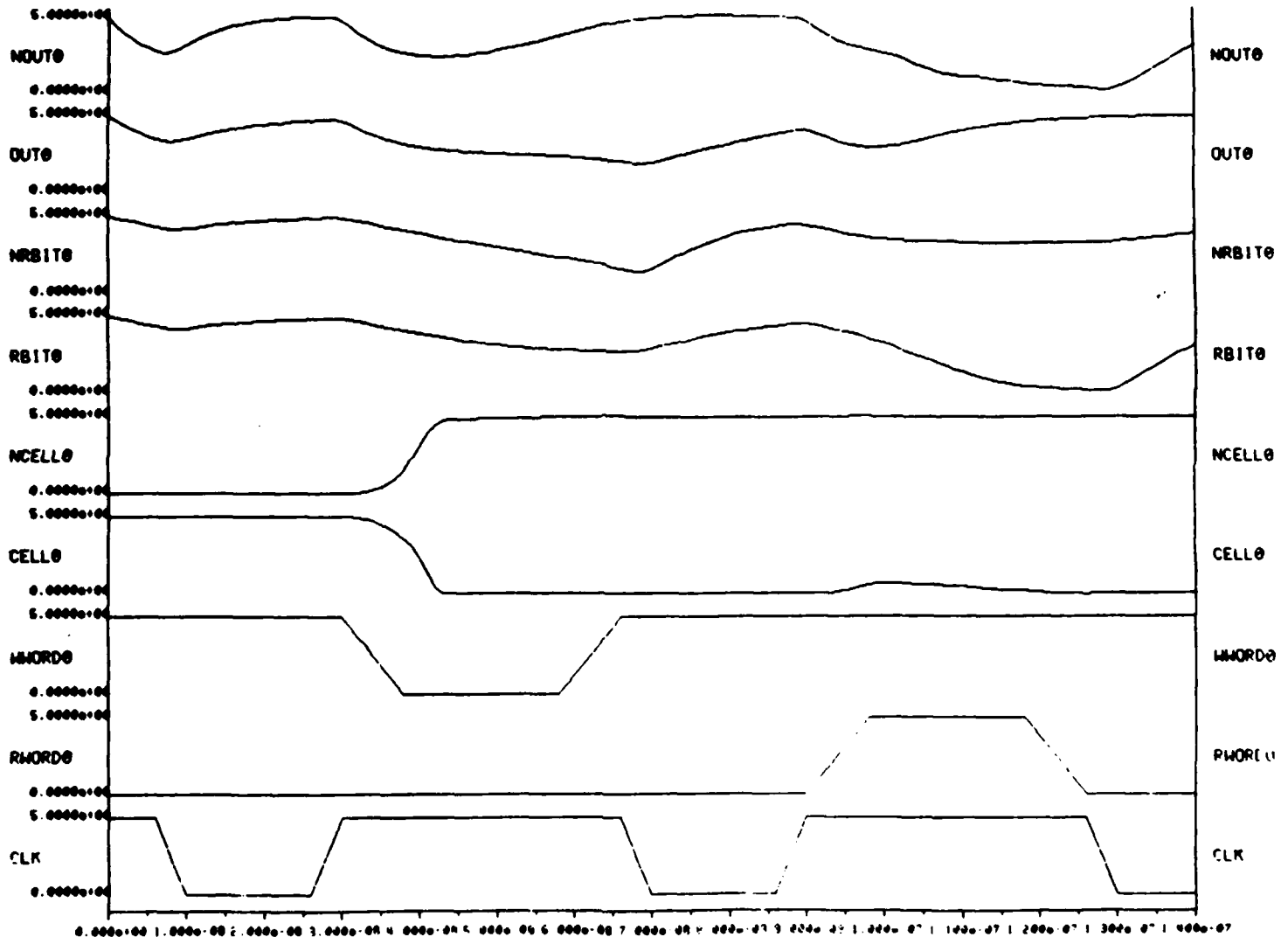


Figure 1: *Spice* Simulation of RAM cell and Sense Amp

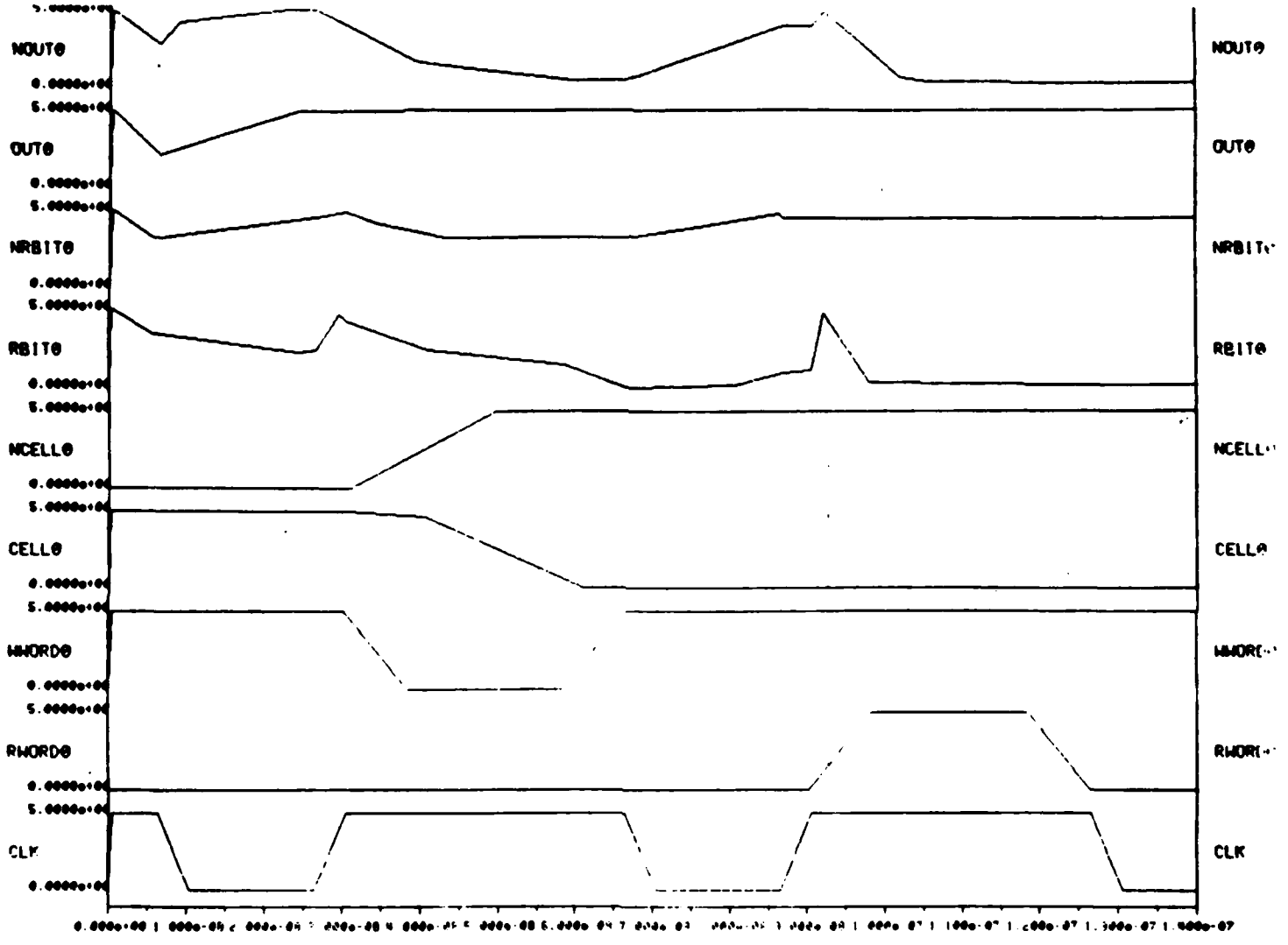


Figure 2: *NC* Simulation of RAM cell and Sense Amp

7 Circuit Designs

7.1 Decoders

(W. Yost)

The decoder generator provides a broad choice of styles of decoder that may be selected. The current generator is an expansion of an earlier version that generated either dynamic nand or nor style decoders using *caesar* format and 3-u CMOS technology. The recent work has used *magic* format and uses the scalable CMOS design rules. At present the following decoder styles are supported:

Style	Format	Technology
1) Dynamic nand	*caesar*	3-u CMOS
2) Dynamic nor	*caesar*	3-u CMOS
3) Dynamic nand	*magic*	scalable CMOS
4) Dynamic nor	*magic*	scalable CMOS
5) Static nand (pseudo-NMOS)	*magic*	scalable CMOS
6) Static nor (pseudo-NMOS)	*magic*	scalable CMOS
7) Gate	*magic*	scalable CMOS

The static nand and nor type decoders have simply had the precharge p-type pullup replaced by a p-type pullup with the gate grounded. The pullup in the nand decoder lengthens with the number of selects in order to maintain the proper gate ratio.

The gate type decoder uses an and gate for each output to decode the select lines. In the case of four or more selects, the selects are split between two nand gates, the outputs of which are put through a nor gate to form the decoder output term. This decoder uses fully static gates.

The generator has the feature that the number of outputs is selectable as a parameter. This allows structures such as 3:6 decoders to be built without waste of area. This also allows an extra select line to be used as an enable on the output. For example in a 4:8 decoder the extra line could be used to disable all outputs.

Portions of this generator have been used in conjunction with the development of a variety of memory elements (see 7.3). Decoders are used with the RAM to decode the row and column addresses. In order to maintain design flexibility when using the

RAM in a chip design it was necessary to have the corresponding flexibility in the decoder outputs. A feature that allows the output decoding to be specified by file input has been developed. This option could be used in cases where only subsets of the full address space are required; no area penalty would be incurred.

7.2 Adders

(W. Yost)

The adder generator creates a Manchester carry adder of arbitrary bit length. In this style adder the carry chain is precharged high, the inputs are examined to determine whether to propagate the carry from the previous bit, generate a carry or to kill any carry from being propagated. The carry that is generated into each bit is then used in the generation of the sum.

This generator utilizes a carry bypass function to speed up the carry evaluation. In this type of structure the carry into a given bit position is also passed several bits in the msb direction. If the inputs to the intervening bits are appropriate, then a gateway is opened for the carry, enabling it to bypass the intervening logic Because the evaluation of whether to open the bypass occurs in parallel across all bits, a speedup over the normal serial passage of the carry occurs.

The carry bypass of this generator is organized into groups of three and four bits. The only exceptions are five bits (organized with a single bypass), two bits (bypass but no intervening bits) and one bit (doesn't need a bypass).

This generator can be selected by proper choice of input parameters to perform the function of subtractor. In addition, the initial carryin bit for an adder can be specified to be tied to the ground rail. The initial carryin to the subtractor can be specified to be tied to the power rail. The default is to have the carryin as an input. This allows nonstandard situations to be handled. An example would be the case of a subtraction in which the signal being subtracted is inverted (active low instead of active high). The desired function could result by performing an addition with the carryin tied high.

7.3 Memory Elements

(W. Barnard)

A family of memory generators has been designed around a common set of sub-modules. Block diagrams of the various memories are found in Figure 1. All memories

utilize identical decoders, precharge circuits, and sense amplifiers. Once the initial single port RAM was designed, the remaining memories were developed at a rate of approximately one every week and a half. This productivity was primarily due to our layout generation tool, *cfl* (Coordinate Free LAP).

Since one of the design objectives was high performance, all of the memories utilize differential signals and a two stage differential sense amplifier. A schematic for this sense amp is given in Figure 2. The resulting layout is 32 lambda by 180 lambda. Particular emphasis was placed on maintaining balanced loading even during mask misalignment. *Spice* simulations indicate the circuit is capable of detecting less than .05PF difference on the BIT lines; memory access times of 8-9 ns (not counting precharge or decoding) have been observed for 100 bit deep memories. Due to the differential nature of the signals, memory access time is relatively independent of memory depth; indeed, for a single bit deep memory the access time is 5-6 ns.

A previously designed decoder generator is used for row decoding and column decoding (See Section 7.1). A dynamic decoder was chosen for performance and area optimization. The decoder precharge cycle overlaps the sense amp BIT line precharge cycle. Thus, the decoder precharge time is included in part of the total memory access time. In the case of the separate I/O RAM, the read port decoder is active high and the write port is active low. In both cases the decoder outputs are all held inactive during precharge.

Two different static memory cells have been designed and are schematically depicted in Figures 3 & 4. In each case the WORD lines are run horizontally in metal1 and the BIT lines are run vertically in metal2 to reduce capacitive load. Power and ground are also routed in metal. The separate I/O RAM of Figure 3 allows simultaneous reading and writing at different clock rates. In the case where the same cell is being read and written, the read port tracks the write port.

The ROM cell is a two transistor implementation with one transistor pulling up and the other pulling down. Since both transistors are p-channel the pull up is not all the way to the power rail. However, the outputs are differential, so fast access times are to be expected. This ROM cell configuration was chosen primarily to fit in with the previously designed peripheral circuitry. A smaller ROM could be designed if ROM-specific peripheral circuits were created.

The CAM is an implementation of a Hopfield Neural Net associative memory model and behaves similarly to a ROM.

The following table lists the memory cell aspect ratios.

cell	aspect ratio in lambda
Neural Net processor element	16 x 16
2 transistor ROM	32 x 21
single port 6 transistor RAM	32 x 41
separate I/O 8 transistor RAM	32 x 61

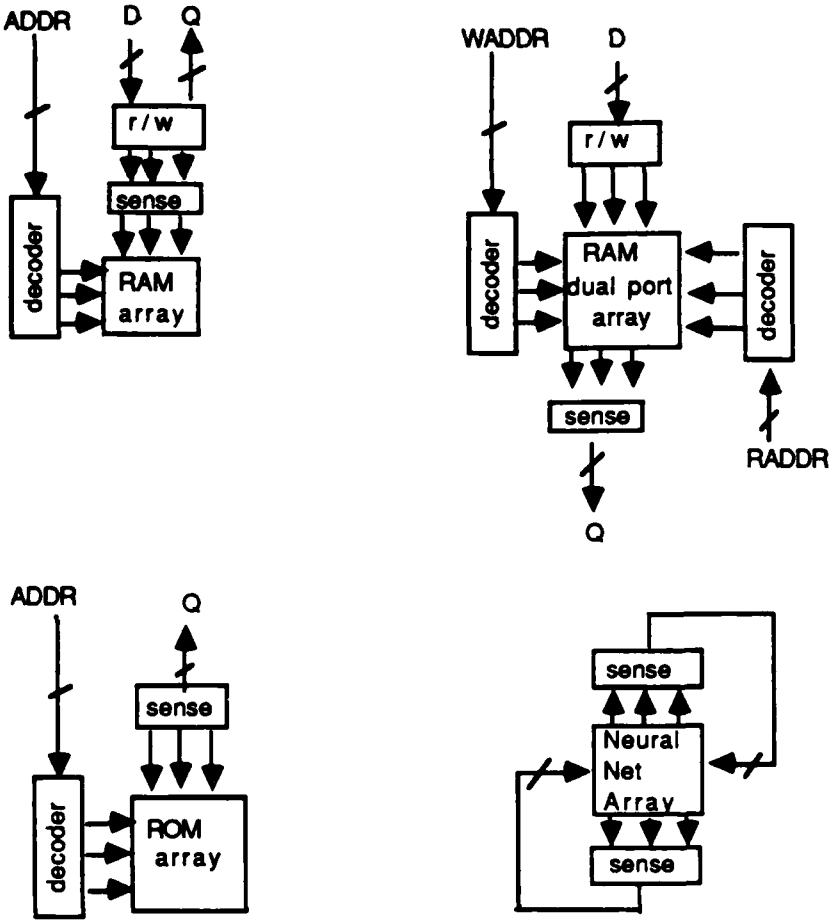

ADDR
D
Q
r/w
sense
decoder
RAM
array
WADDR
D
r/w
RAM
dual port
array
decoder
decoder
sense
RADDR
Q
ADDR
Q
sense
decoder
ROM
array
sense
Neural
Net
Array
sense

Figure 1

Two Stage SENSE AMP

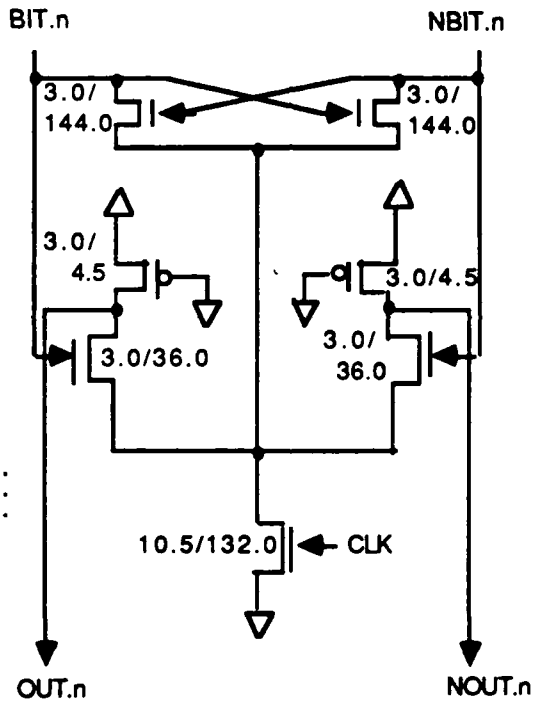


Figure 2

Single Port RAM CELL

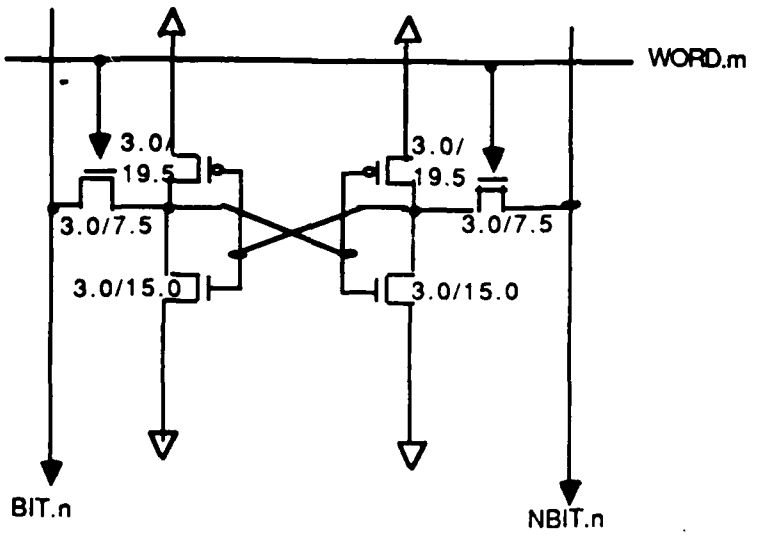


Figure 3

Separate I/O RAM CELL

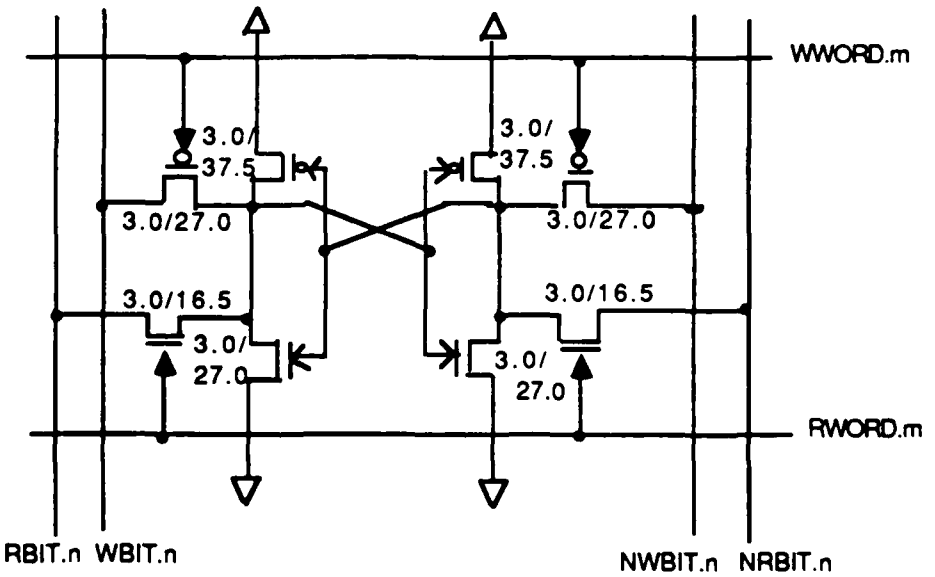


Figure 4

Appendix A

EBNF of the Declarative Description MN

In the definition given below, the equal sign, "=", is to be read as "is defined as". All character literals are enclosed in double quotes (i.e., ""). Integer literals are represented as ILIT. String literals are represented as SLIT. A letter followed by an optional number of letters, characters or "_", in any order is represented as an ID. A double quoted, upper case character string is a keyword. The vertical bar "|" is used to separate alternatives in the definitions. Braces (i.e. " { " and " } ") specify zero or more repititions. Brackets (i.e. " [" and "] ") express option. Each definition ends with a " . ".

```
<program> = <declaration> "{" <statement> [ ";" <in_spec> ]
{ ";" <statement> } "}".

<declaration> = <name_decl> <type_decl> <param_decl> [ <cell_decl> ]
[ <func_decl> ].

<name_decl> = "NAME" ID ";".

<type_decl> = "TYPE" <type_group> ";".

<type_group> = "LAYOUT" | "SCHEMATIC" | "FUNCTIONAL".

<param_decl> = "PARAMETER" <param> { "," <param> } ";".

<param> = ID "=" ILIT | ID "=" SLIT.

<cell_decl> = "LEAF" "CELLS" ID { "," ID } ";".

<func_decl> = "FUNC" ID { "," ID } ";".

<statement> = <regular_statement> | <out_spec>.

<regular_statment> = <object> "=" <body> { "=" <body> }.
```

```
<object> = ID [ "[" <index_list> "]" ].

<index_list> = <expr> { "," <expr> }.

<expr> = <expr> <op> <expr> | <term>.

<op> = "**" | "/" | "%" | "+" | "-" | "*" |
"&" | "^" | "|" | "|^" | "--" | "--^".

<term> = ILIT | SLIT | <object> | <function> | ["-"|"~"] <term> |
"(" <expr> ")" | "(" <loop> ")" | FILL "(" <expr> ")" |
<geo_op> "(" <object> ")" | "ROT" "(" <object> "," <expr> ")".

<function> = ID "(" <func_parms> ")".

<func_parms> = <expr> [ "," <index_list> | "," <subrange> [ "," <expr> ] ].

<subrange> = ID "=" <expr> ".." <expr>.

<geo_op> = "MX" | "MY".

<body> = <assignment> [ "," <IfCond> ].

<assignment> = <expr> | <out_assign>.

<out_assign> = "OUTPUT" <in_out>.

<in_out> = "[" ID "]" "(" subrange [ "," <expr>] ")".

<in_spec> = "INPUT" "=" <input> { "," <input> }.

<input> = ID <in_out>.

<out_spec> = "OUTPUT" [ "[" <index_list> "]" ] "=" <out_body>
{ "=" <out_body>}.

<out_body> = <assignment> ":" <expr> [ "," <IfCond> ].

<loop> = <loop_op> "(" <object> "(" <loop_index> ")" ")".
```

```
<loop_op> = "+" | "*" | "&" | "^" | "|" | "|^" | "--" | "--^".

<loop_index> = <expr> | <subrange> [ "," <expr> ].

<IfCond> = "IF" <relation> { "&&" <relation> | "||" <relation> }.

<relation> = <expr> <re_op> <expr>.

<re_op> = "<=" | "<" | "==" | "!=" | ">" | ">=".
```

Notes on Semantics:

Parameters may be integers or strings. Terms may be integers, strings or geometric objects, but must make sense in context. For example,

$$A = 2 + "xyz";$$

is meaningless. The geometric operators are vertical and horizontal juxtapose ("|" and "—", respectively) and vertical and horizontal juxtapose with overlap ("|^" and "—^", respectively). Loops must evaluate a positive number of times (i.e. no "NULL" result is allowed in loops). The "FILL" term only has meaning when composing it with a geometric operand.

Appendix B

Application of MN to a Decoder

(W. Yost)

The gate type decoder is presented here as an example of the use of Meei's Notation (MN) for creating schematic and layout diagrams. The decoder is a commonly used logic block in which "n" selects are decoded causing one of 2**n (or less) outputs to go active.

In this example the "inv" parameter controls the sense of the output. This decoder is normally (inv=0) active high. The "h" parameter controls the number of output terms in the decoder. This must be <= 2**(#selects). The "m" and "n" parameters determine the number of selects. In the gate decoder each output term is derived from the selects feeding an "m" input nand gate and an "n" input nand gate which in turn feed a nor gate. The "m" and "n" parameters would normally be equal or within one of each other. Thus (m+n) represents the total number of select lines.

The MN for the layout is shown in Figure 1. The architecture splits the circuit into a pullup plane and a pulldown plane. This is indicated in the first line of programming. Other structures include a row of input buffers that fits onto the bottom of each half and the output nor gates that are stuck onto the right hand side of each decoder row. Figure 2 shows the tiling that is performed in this example while Figure 3 shows the expanded *Magic* format layout. The MN for the schematic representation is shown in Figure 4. The schematic itself is shown in Figure 5.

A comparison of the notation for the layout and schematic forms shows almost a line for line correspondence. As much as possible cells that were used in a common function were given the same names. The last half of the schematic notation performs some routing of common inputs that isn't done in the layout. Other than that the major differences are due to the fact that the schematic notation performs labeling, a function that for the layout is performed outside of the context of the notation.

The rest of the differences came about through the simplification that can take place in the schematic. Schematic glue pieces are generally not required. In the schematic gdec_empt[1 2 ...] cells were used as spacers at points where cells routing power and ground were used in the layout. Some pieces of layout (gdec_highr — gdec_lowj) were joined into a single schematic cell (gdec_npjoin). One other difference is that in the schematic a single function Z[ ] filled the function of Y[ ] and Z[ ] in layout. A single

function required two slightly different layouts. In the schematic this was not so and a single cell sufficed.

Other than the labeling, the differences between notations could be eliminated if desired by simply not taking advantage of the minor simplifications that can be made in the schematic notation. By judicious partitioning of the schematic leaf cells and maintenance of a naming scheme the two notations could be made almost interchangeable.

```
NAME gdec;
TYPE LAYOUT;
PARAMETER
inv=0,
h=6,
m=2,
n=2;

LEAF CELLS
gdec_lowj, gdec_lr, gdec_lr1, gdec_ninv_fill, gdec_njoin, gdec_nor, gdec_o_inv,
gdec_one, gdec_one1, gdec_pmr, gdec_pone, gdec_high, gdec_pone1, gdec_highr,
gdec_pzero, gdec_i_inv, gdec_pzero1, gdec_i_ninv, gdec_route1,
gdec_inv, gdec_route2, gdec_inv_fill, gdec_route3, gdec_inv_fill1,
gdec_zero, gdec_low, gdec_zero1, gdec_11, gdec_route4;

FUNC binary;

{
    gdec     = left -- right;

    left     = left_bot | (|(left_row[i](i=h-1..0)));
    left_bot= gdec_11 -- (--(gdec_i_inv(m+n))) -- gdec_inv_fill;
    left_row[i]  = gdec_high -- (--( W[i,j](j=m+n..n+1))) --
                        (--( X[i,k](k=n..1))) -- gdec_highr -- gdec_lowj;
    W[i,j] = gdec_pzero, IF binary(i,j) == 1
           = gdec_pone;
    X[i,j] = gdec_pzero1, IF binary(i,j) == 1
           = gdec_pone1;

    right     = right_bot | (|(right_row[i](i=h-1..0)));
    right_bot= (--(gdec_i_ninv(m))) -- gdec_ninv_fill --
                        (--(gdec_i_ninv(n))) -- dec_botend;
    dec_botend = gdec_lr, IF inv==0
               = gdec_lr -- gdec_lr1;
    right_row[i] = (--( Y[i,j](j=m+n..n+1))) -- gdec_njoin
                 --( --( Z[i,k](k=n..1))) -- dec_end;
    dec_end = gdec_nor, IF inv==0
            = gdec_nor -- gdec_o_inv;
    Y[i,j] = gdec_one, IF binary(i,j) == 1
           = gdec_zero;
    Z[i,j] = gdec_one1, IF binary(i,j) == 1
           = gdec_zero1;

}
```

Figure 1

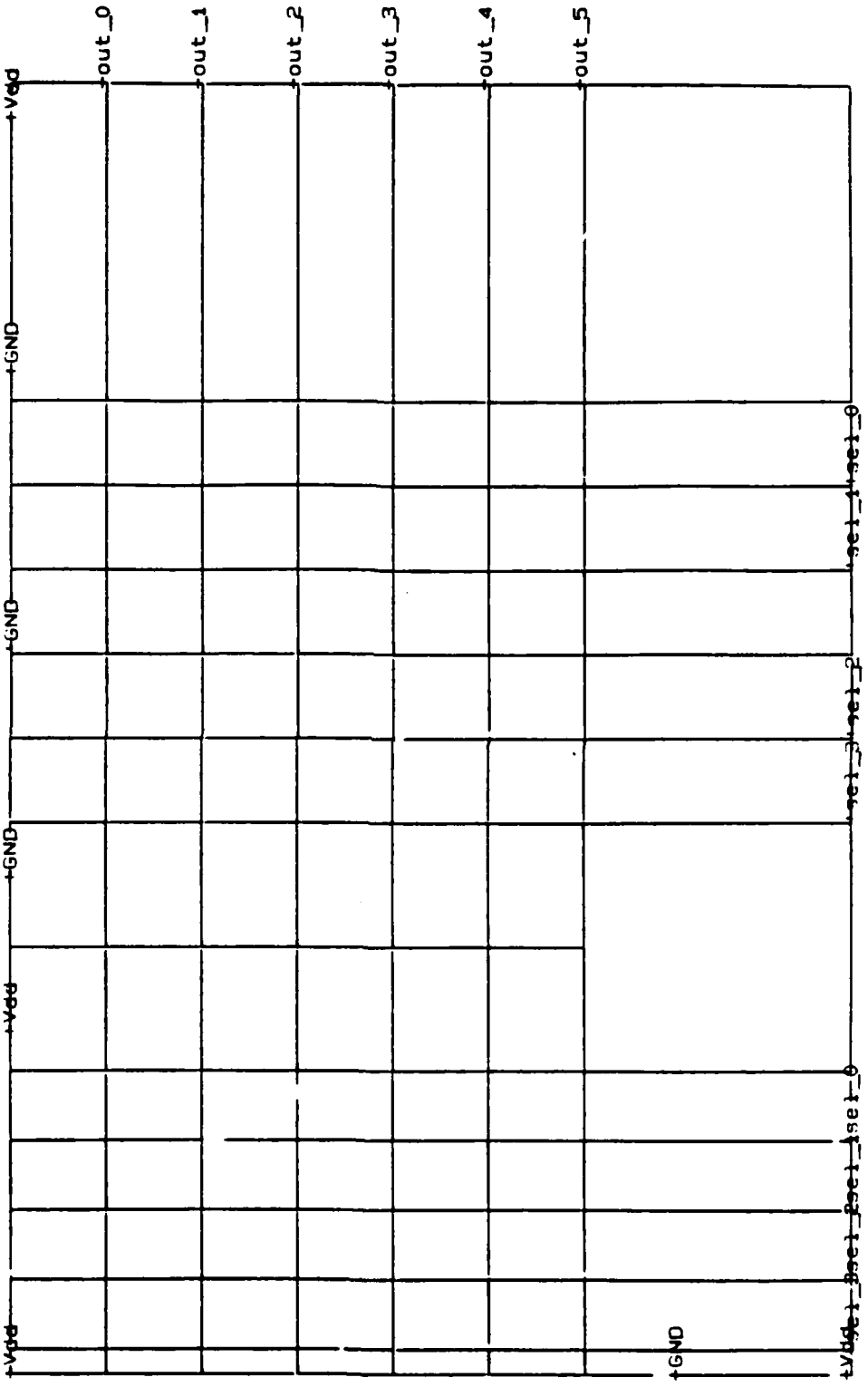

out_0
out_1
out_2
out_3
out_4
out_5
Vdd
GND
GND
GND
Vdd
Vdd
GND
set_0
set_1
set_2
set_3
set_4
set_5

Figure 2

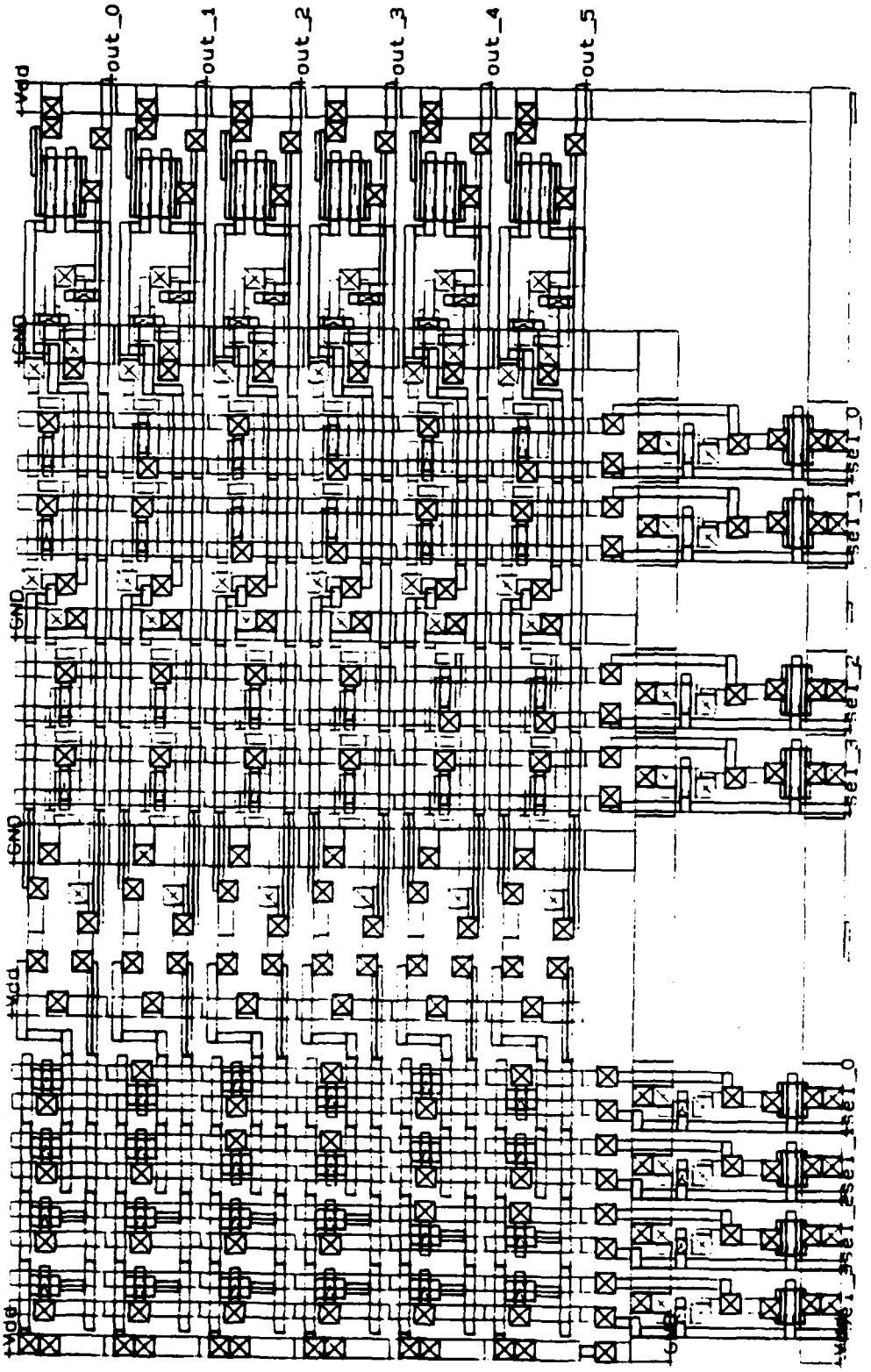
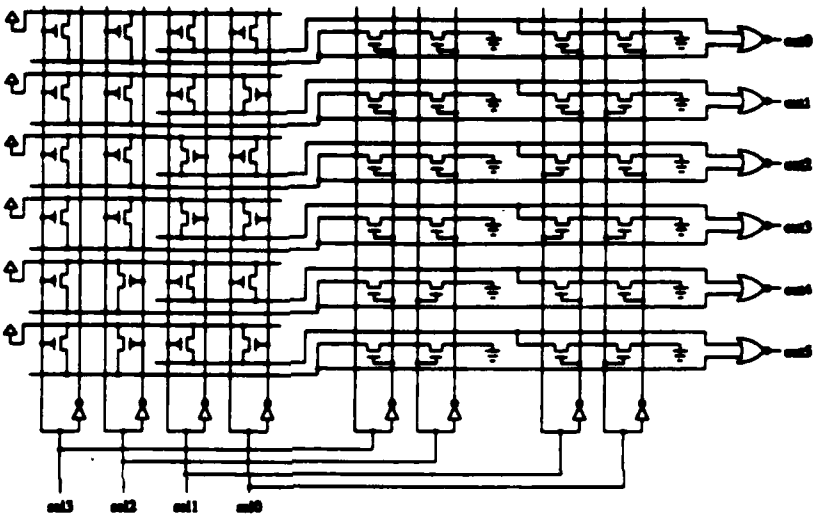


Figure 3

```
NAME decoder;
TYPE SCHEMATIC;
PARAMETER
inv=0,h=6,m=2,n=2;

LEAF CELLS
gdec_empt1,gdec_empt2,gdec_empt3,gdec_empt4,gdec_i_inv,
gdec_nor,gdec_nend,gdec_njoin,gdec_npjoin,gdec_o_inv,
gdec_one,gdec_pone,gdec_pone1,gdec_pzero,gdec_pzero1,
gdec_route1,gdec_route2,gdec_route3,gdec_route4,gdec_route5,
gdec_high,gdec_zero;

FUNC binary, strcat, int2str;

MAIN {
    decoder = in_lines | body;
    body    = left -- right;
    in_lines= connectors--corners;

    left    = left_bot | (|(left_row[i](i=h-1..0)));
    left_bot= gdec_empt1-- (--(gdec_i_inv(m+n))) -- gdec_empt2;
    left_row[i]  = gdec_high-- (--( W[i,j](j=m+n..n+1))) --
                        (--( X[i,k](k=n..1))) -- gdec_npjoin;
    W[i,j] = gdec_pzero, IF binary(i,j) == 1
           = gdec_pone;
    X[i,j] = gdec_pzero1, IF binary(i,j) == 1
           = gdec_pone1;

    right    = right_bot | (|(right_row[i](i=h-1..0)));
    right_bot= (--(gdec_i_inv(m))) --gdec_empt2--
                        (--(gdec_i_inv(n)));
    right_row[i] = (--( Z[i,j](j=m+n..n+1))) --gdec_njoin
                        --( --( Z[i,k](k=n..1))) --gdec_nend--dec_end[i];
    dec_end[i] = gdec_nor[strcat("out", NDX[i])], IF inv==0
           = gdec_nor[""] -- gdec_o_inv[strcat("out", NDX[i])];
    NDX[i]  = int2str(48 + i),                    IF i < 10
           = strcat( int2str(48 + i/10), int2str(48 + i%10)),
                        IF i >= 10 && i < 100
           = "***";
    Z[i,j] = gdec_one, IF binary(i,j) == 1
           = gdec_zero;

    connectors = (|(conn_row[i](i=m+n..1)));
    conn_row[i]= gdec_empt3-- (--(conn_element[i,j](j=1..m+n)))
                        --gdec_route5;
    conn_element[i,j]
        = gdec_route1[""], IF i == j && i != (m+n)
        = gdec_route1[strcat("sel",NDX[m+n-j])], IF i==j && i==(m+n)
        = gdec_route4[""], IF i>j && i!=(m+n)
        = gdec_route4[strcat("sel", NDX[m+n-j])], IF i>j && i==(m+n)
        = gdec_route3;

    corners     = corner1 | corner2;
    corner1     = (|(corn_row[i](i=m+n..m+1)));
    corner2     = (|(corn_row1[i](i=m..1)));
    corn_row[i]= (--(corn_element[i,j](j=1..m))) --gdec_route5
                        -- (--(corn_element[i,k](k=m+1..m+n)));
    corn_row1[i]= (--(corn_element[i,j](j=1..m))) --gdec_empt4
                        -- (--(corn_element[i,k](k=m+1..m+n)));
    corn_element[i,j]
                = gdec_route2, IF i==j
                = gdec_route5, IF i>j
                = gdec_route4;

}
```

Figure 4

Figure 5



Appendix C

Application of MN to a Multiplier

(M. Bailey)

A multiplier is presented here to illustrate the use of the generator back ends. This multiplier is unsigned, with variable multiplicand and multiplier widths. The minimum width supported for the multiplicand and multiplier is 3. This multiplier has sizable input drivers, with independent driver strengths available for the multiplier and the multiplicand. Also, portions of the GND and Vdd busses can be independently sized.

The "Meei's Notation"(MN) for the multiplier layout is shown in Figure 1 and Figure 2. Note that the parameter list contains entries for the variables listed above. The "m" parameter controls the width and the "n" parameter controls the height of the multiplier array. These specify the widths of the multiplier and the multiplicand.

The multiplier is composed of five major structures. These are:

- Multiplier plane (mplane)
- Drivers for the multiplier (xdriver)
- Drivers for the multiplicand (ydriver)
- Ripple adder for the result (adder)
- Ground bus and low-order output bits (rside)

The positioning of these structures is described in the notation. Note that the ydrivers overlap the mplane. The adder is overlapped with the multiplier plane and drivers so that the Vdd bus at the bottom of the multiplier plane can be overlapped with the Vdd bus at the top of the adder array. All of the other top-level cells are tiled.

In the definition of the adder, the leftadd cell is described as overlapping the addrow cell even though there is really no overlap desired. In fact, the overlap operator is used to align the cells so that the Vdd busses are connected.

In the definition of the xdriver cells, notice that xbuf differs depending on the widths of the Vdd and GND busses. This is described in the notation by using the IF

construct. In this case there are four different possibilities, depending on the sizes of the busses. The cases are order dependent and are evaluated in the order listed.

The resulting layout is shown in Figure 3 and Figure 4. Figure 3 shows the cells as they are constructed using MN, and Figure 4 is the expanded layout.

The MN for the multiplier schematic is shown in figure 5. The highest level definition of the multiplier is similar to that in the layout description in that they are both composed of the five structures listed above. The only difference is that there are no overlapping operators in the schematic description, and thus all overlapping operators in the layout description have been replaced with the corresponding non-overlapping operator.

It is in the sub-definitions that the two descriptions differ. While the layout description contains complete information on the multiplier, the schematic description of the multiplier (Figure 6) shows function blocks for the adder cells and the multiplier cells. Since the schematic is a documentation tool, the level of detail shown in a view is based on clarity. Thus the MN for the schematic is simpler than that of the layout.

Because the schematics of the function blocks in the schematic are of interest to the user, these schematics are provided as separate schematics. There is one schematic for the full multiplier (Figure 7), and another for the adder (Figure 8). Since these schematics are static descriptions of cells, it is not necessary for these cells to use MN. However, these cells were generated using MN because it was easy.

```
NAME mult;
TYPE LAYOUT;
PARAMETER m = 4, n = 4, xdrive = 7, ydrive = 25, vdd_bus = 20,
        gnd_bus = 30;
LEAF CELLS
          maul, mam, maur, mav, may,
          mya, myf, myg, myo, myb, mybs,
          mulf, mulv, mulh,
          mmd, mmg, mmu, mmv, mmbv,
          mxa, mxf, mxg, mxo, mxv, mxgd,
          mra, mrab, mrb, mrs, mrst, mrtb,
          mrsb, mrbb, mrout, mro, mrx, mrc;

{

    mult = (adder |^ (ydriver --^ (mplane | xdriver ))) -- rside;

    adder =leftadd --^ addrow;

            addrow =  maul -- (-- (mam(m-1))) -- maur;
            leftadd = addvbus --^ addxtnd;
            addvbus = mav, IF vdd_bus <=7
                    = (--(mav((vdd_bus-1)/7 + 1)));
            addxtnd = may, IF ydrive <= 7
                    = (--(may((ydrive-1)/7 + 1)));

    ydriver = yalmost |^ upleft, IF vdd_bus <= 7
            = yvbustub |^ yalmost |^ upleft;

            yalmost = (|^(ybuf(i=1..n)));
            ybuf = yvbus --^ (mya --^ ytran2 --^ myo), IF vdd_bus > 7
                 = myb --^ (mya --^ ytran2 --^ myo);
            ytran2 = myf, IF ydrive <= 7
                   = myg --^ (--^(myf((ydrive-1)/7)));
            yvbus  = (--(myb((vdd_bus-1)/7 + 1)));
            yvbustub  = (--(mybs((vdd_bus-1)/7)));

            upleft = vertv, IF vdd_bus <= 7
                   = horizv |^ vertv;
            horizv = (|(horizrow((vdd_bus-1)/7)));
            horizrow = (--(mulf((vdd_bus-1)/7 + 1 + (ydrive-1)/7))) -- mulh;
            vertv = (--(vertrow((vdd_bus-1)/7 + 1)));
            vertrow = mulv , IF xdrive<= 7 && gnd_bus <=7
                    = (|(mulf((xdrive-1)/7))) | mulv, IF gnd_bus <= 7
                    = mulv | (|(mulf((gnd_bus-1)/7))), IF xdrive <= 7
                    = (|(mulf((xdrive-1)/7))) | mulv |
                        (|(mulf((gnd_bus-1)/7)));
```

Figure 1: MN for the Multiplier Layout

```
mplane = (|(row[i](i=1..n)));

        row[1] = bot_left --^ (--^(bot(m-1)));
        row[a] = mid_left --^ (--^(plain(m-1)));
        bot_left = b_plain_plus;
        bot      = b_plain_plus;
        mid_left = plain;
        plain_plus = mmv | plain;
        plain      = mmd | mmg | mmu | mmv;
        b_plain_plus = mmbv | plain;

xdriver = (--^(xbuf(i=1..m)));

        xbuf = mxo |^ xtran2 |^ mxa, IF vdd_bus <= 7 && gnd_bus <= 7
             = xvbus | (mxo |^ xtran2 |^ mxa), IF gnd_bus <= 7
             = (mxo |^ xtran2 |^ mxa) | xgbus, IF vdd_bus <= 7
             = xvbus | (mxo |^ xtran2 |^ mxa) | xgbus;
        xvbus = (|(mxv((vdd_bus-1)/7)));
        xgbus = (|(mxgd((gnd_bus-1)/7)));
        xtran2 = mxf, IF xdrive <= 7
               = (|^(mxf((xdrive-1)/7))) |^ mxg;

rside = addgnd |^ gndcol |^ topgndcol;

        addgnd = mra, IF gnd_bus <= 7
               = mra -- (--(mrab((gnd_bus-1)/7)));
        gndcol = (|^(gndout[i](i=1..n)));
        gndout[n] = mrst, IF gnd_bus <= 7
                  = mrst -- (--(mrtb((gnd_bus-1)/7)));
        gndout[1] = mrsb --^ mrout, IF gnd_bus <= 7
                  = mrsb -- (--(mrbb((gnd_bus-1)/7))) --^ mrout;
        gndout[a] = mrs --^ mrout, IF gnd_bus <= 7
                  = mrs -- (--(mrb((gnd_bus-1)/7))) --^ mrout;

        topgndcol =  mrxrow |^ gcfillrow, IF vdd_bus <= 7 && xdrive <= 7
                  = (mrxrow | xfillrow) |^ gcfillrow, IF vdd_bus <= 7
                  = (vfillrow | mrxrow) |^ gcfillrow, IF xdrive <= 7
                  = (vfillrow | mrxrow | xfillrow) |^ gcfillrow;
        vfillrow = (--(vfill((gnd_bus-1)/7 + 1)));
        vfill = (|(mro((vdd_bus-1)/7)));
        xfillrow = (--(xfill((gnd_bus-1)/7 + 1)));
        xfill = (|(mro((xdrive-1)/7)));
        gcfillrow = (|(gcfill((gnd_bus-1)/7 + 1)));
        gcfill = mrc, IF gnd_bus <= 7
               = mrc -- (--(mro((gnd_bus-1)/7)));
        mrxrow = (--(mrx((gnd_bus-1)/7 + 1)));

}
```

Figure 2: MN for the Multiplier Layout (continued)

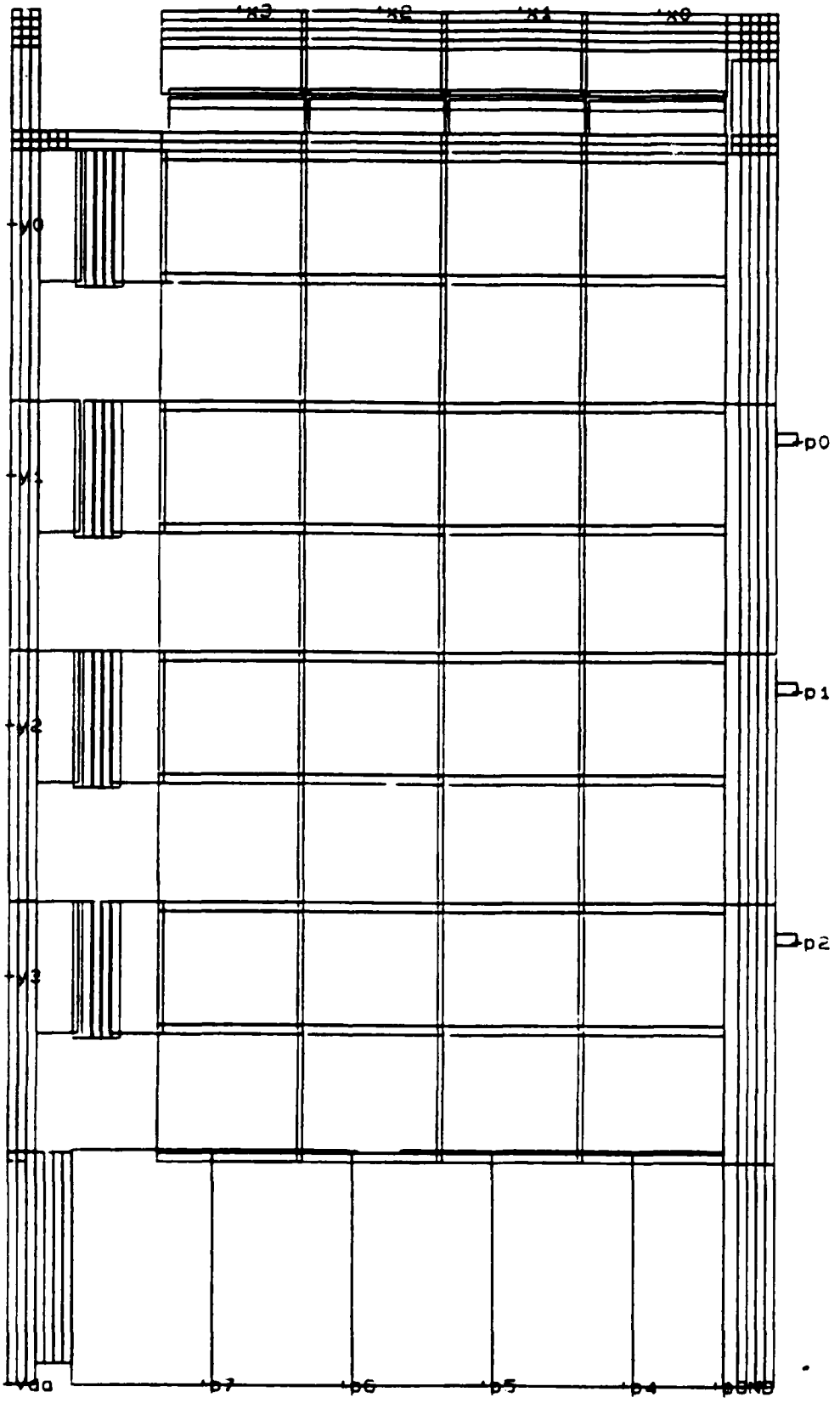


Figure 3: Cell Layout for the Multiplier

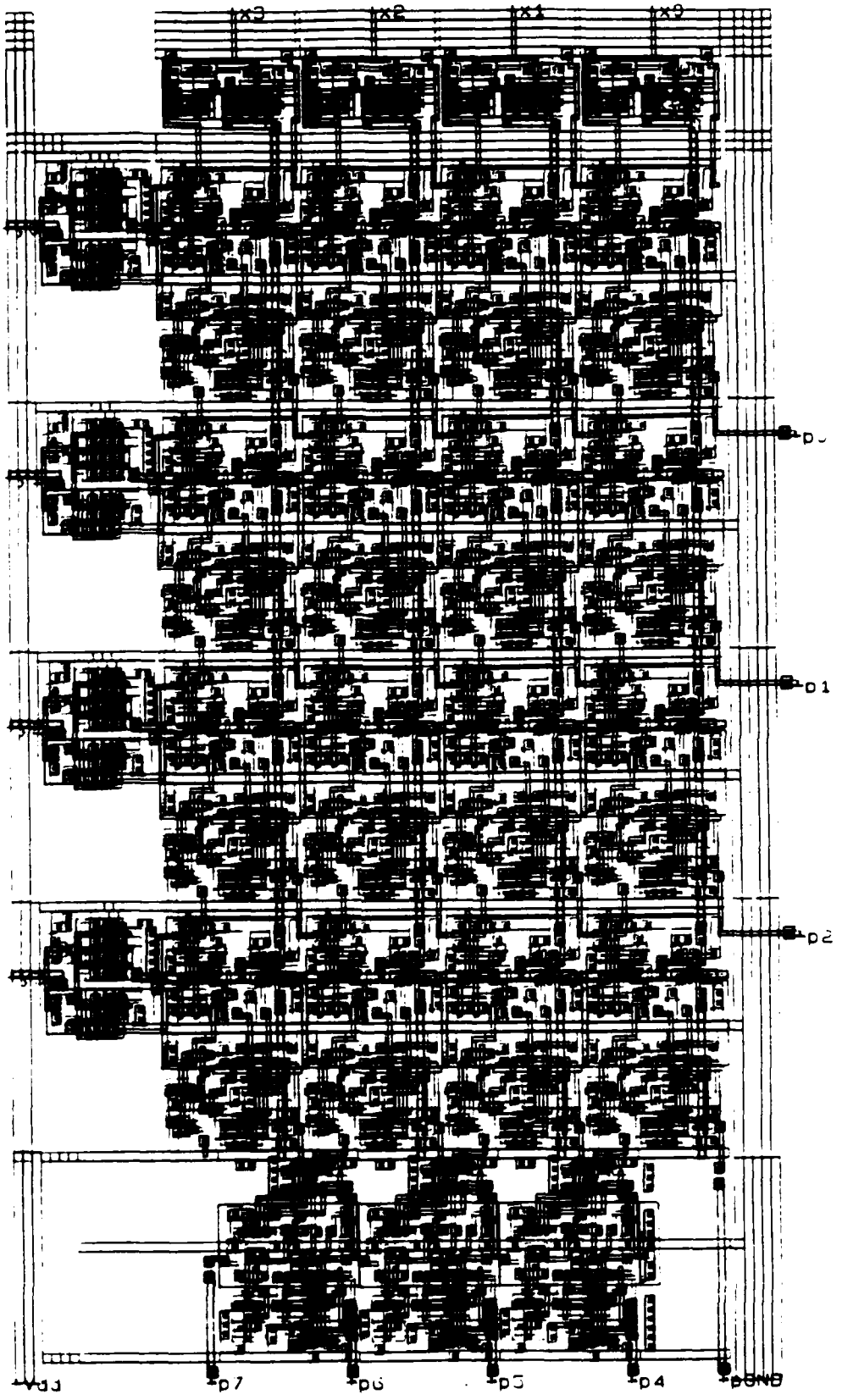


Figure 4: Expanded Cell Layout for the Multiplier

```
NAME mult;
TYPE SCHEMATIC;
PARAMETER  m = 4, n = 4;
LEAF CELLS fullmult_sch, xbuf_sch, rxbuf_sch, ybuf_sch, bybuf_sch,
                topcorner_sch, empty_sch, brside_sch, rside_sch,
                ladd_sch, add_sch, radd_sch,
FUNC strcat, int2str;

{
        mult = (adder | (ydriver -- (mplane | xdriver))) -- rside;

        adder = empty_sch -- empty_sch --
                 ladd_sch[strcat("p",int2str(n+1)),strcat("p",int2str(n+2))]
                 -- radd_sch["0","ca",strcat("p",int2str(n))], IF m <= 3
              = empty_sch -- empty_sch --
                ladd_sch[strcat("p",int2str(m+n-2)),strcat("p",int2str(m+n-1))]
                 -- (--(add_sch["ca",strcat("p",int2str(n+j))](j=1..m-3)))
                 -- radd_sch["0","ca",strcat("p",int2str(n))];

        ydriver =  bybuf_sch[strcat("y",int2str(n-1))] |
                   (|(ybuf_sch[strcat("y",int2str(n-i)),"0"](i = 2..n)))
                   | topcorner_sch["0"];

        mplane = (|(row[i](i=1..n)));
                row[i] = (--(fullmult_sch["c",strcat("x",int2str(m-j)),
                        "s",strcat("y",int2str(n-i))](j = 1..m)));

        xdriver = (--(xbuf_sch["0","0",strcat("x",int2str(m-j))](j = 1..m-1)))
                  -- rxbuf_sch["0","x0"];

        rside = brside_sch[strcat("p",int2str(n-1))] |
                (|(rside_sch[strcat("p",int2str(n-i))](i=2..n)))
                | empty_sch | empty_sch;

}
```

Figure 5: MN for the Multiplier Schematic

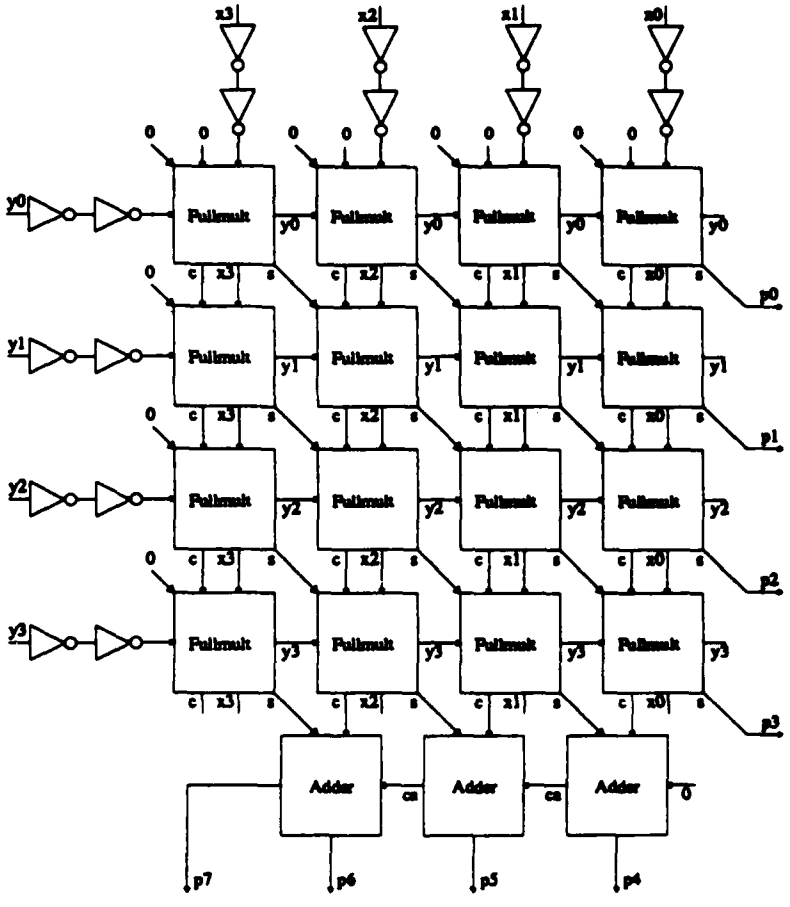


Figure 6: Schematic Diagram for the Multiplier

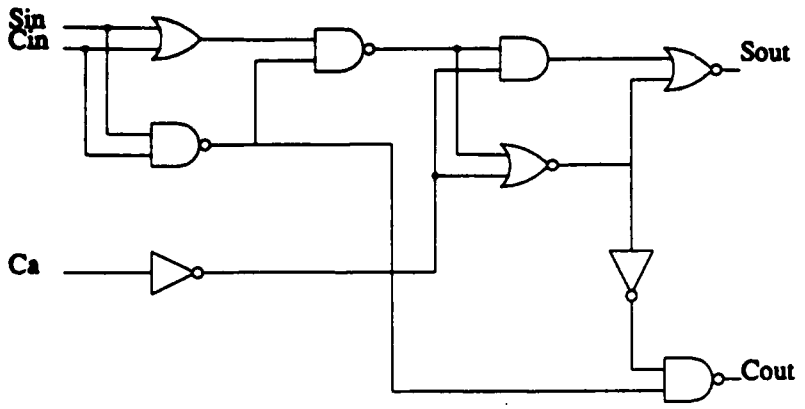


Figure 7: Schematic Diagram for the FullMult Cell

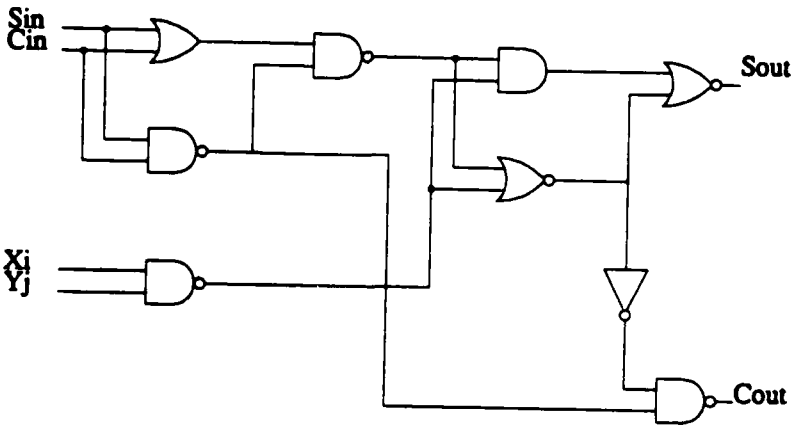


Figure 8: Schematic Diagram for the Adder Cell

Appendix D

Hercules: A Power Analyzer for MOS VLSI Circuits[1]

(Extended abstract submitted for publication)

AKHILESH TYAGI
FR-35, Department of Computer Science
University of Washington
Seattle, WA 98195

ABSTRACT

Hercules is a *stage* based MOS power estimator. The present CMOS version reports the average and peak load current & the average and peak direct current due to slow rising input signals for each stage activated by a clock or input transition. A tree like data structure models the Vdd and ground distribution from the pin to sources/drains. The voltage drops from the pin to sources/drains are also reported. Note that an extension of *Hercules* to handle nMOS circuits is also planned. It is relatively straightforward to extend this algorithm to estimate the DC power consumption.

[1]Supported in part by DARPA under Contract MDA903-85-K-0072

1 Overview

The contributions of this paper are as follows.

- We give a linear average time algorithm for computing current levels based on *stage* decomposition [Ous85] of a CMOS VLSI circuit. A *stage* is a chain of switches followed by either an output or a gate. The stages are traced out in a depth-first order enabling us to deal with the cross-coupled memory elements. The current implementation is an extension of a timing verification program, *crystal* [Ous83]. Hence, it supports all the mechanisms of *crystal* for flow specification.

- We have an accurate switch-level model of short-circuit current in CMOS inverters and static circuits. In CMOS circuits, slow input signal edges give rise to short-circuit current from *Vdd* to *ground*. The duration and magnitude of this current depend on the input signal slope, load and the device gain. Interestingly, all these factors can be encapsulated into a single number, *rise time ratio*, as observed by Ousterhout [Ous84] in a different context. The *rise time ratio* is the ratio of input signal slope to the native output signal speed. Most of the digital circuits are designed with only a fixed set of load and transistor sizes. The information about these structures can be extracted from SPICE runs on basic types of devices occuring in the circuit. Thus this model can predict the short-circuit current levels to within 20% of SPICE calculations. Ousterhout [Ous84] was first to use this fact to model the effective resistances of devices in *crystal*.

- As the plot in Figure 2 shows, the total charge flow is minimum at the rise time ratio of one, or when the input and output signal have the same rise and fall times. Thus the closer the rise time ratio of a stage is to 1, the more optimal is the driver sizing with respect to power consumption. *Hercules* can report all the stages that have a rise time ratio larger than a user specified threshold. Clearly, this information is very useful in sizing cascaded drivers.

- We extend the metal tree idea of Wilson [Wil85] to accomodate multilayer metal and loops. Typically, a VLSI circuit does not have many loops in its power distribution metal bus structure. We found that the comb structure, as shown in Figure 3, was the most common form of loop encountered in power buses. Notice that it is a very compact biconnected component, in graph theoretic terminology. We are able to deal with them very efficiently using a depth-first based search technique.

All these components are embedded in the power analysis program *Hercules*. The current version of this program does not incorporate the metal tree structure. We have a separate program to construct a 2-layer multipin metal tree. We are working on a program for dealing with the loops (connected components).

2 Introduction

With the increasing complexity of integrated circuits, came a wide gamut of tools to assist a designer. Graphics layout systems, geometric design rule checkers, circuit extractors, simulation tools and timing analyzers are only few examples of the tools available to manage the mass complexity of the chips. One area which is not represented in this list is power analysis. In a large chip it is difficult to keep track of current requirements of different components. Consequently, many a time, the buses may not be sized properly resulting in *metal migration* and noise problems. Although, the power consumption of a CMOS device is dramatically smaller than that of an nMOS device, the size of current day chips has made it necessary to pay more attention to power requirements of a CMOS chip. Most often in a high performance chip, the drivers are sized up to provide small delay times. Pipelined systems are very commonly used in the high performance architectures. For such systems most of the area is switched during each clock cycle [ST86]. In addition, the power consumption is a function of the frequency of operation. Thus the problem is even more critical for the high performance systems.

Especially important is the information derived from the voltage drops from the *power (ground)* pin to the drain (source) terminal of the transistors. For a brief duration, when the peak transient currents are very high, this drop can be significant. The operating voltage span of a logic device falls sharply during this time. Sometimes the devices are not designed with this much noise tolerance. The pin to device voltage drops provide very useful diagnostic information about a chip. The average current levels are used to size the power buses to avoid metal migration problems.

RELATED WORK: Several tools exist for the power analysis of nMOS circuits. The Berkeley tools included a power estimator called *powest* for nMOS [Cme82]. It counts the number of pull-up load devices (*e.g.* depletion load). It makes assumptions about the fraction of time each of these load devices can be *on*. More recently, Jeff Wilson developed *pwranal* to estimate the power requirements of an nMOS chip [Wil85]. He also developed the idea of *metal tree*. *pwranal* uses the metal tree representation to report the pin to device terminal voltage drops. However, *pwranal* can deal with only a single power pin and one layer metal. Moreover it breaks the metal bus structure loops arbitrarily.

3 Current Estimation Algorithm

We wanted to be able to model the current flow as an instance of *max flow* problem with charge being the commodity pushed through the network. The switch model provides the ideal starting point for this idea. Each switch can be thought of as a node in the commodity network with the switch capacity (transistor width) and switch status (*on* or *off*) determining the node capacity. Applying this idea to the whole circuit can still be expensive. For an exact analysis, we need to

The stage activated by signal in1 going from 0 → 1 is shown by the thick line. It starts with the n-chanel connecting to in1, followed by two pass transistors A and B, terminating in the gate C.

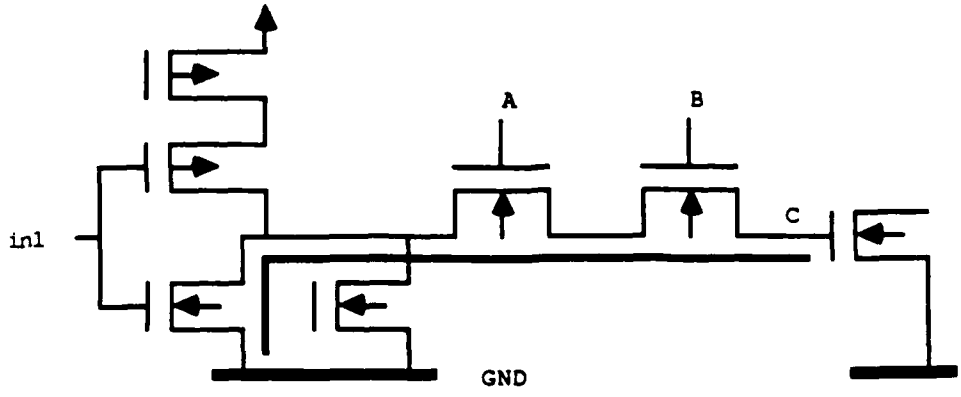


Figure 1: An Example of a Stage

determine the state of all the switches in the network to work out the node capacities. This involves a complete simulation of the circuit for all 2^n input vectors (for an n input circuit), to determine the worst case. Thus, we need to take a value-independent approach to control the problem complexity. like the timing verifiers TV [Jou83] and Crystal [Ous85] do. The *stage* decomposition provides just the right abstraction for an approximate analysis. A *stage* is a chain of transistors leading from a strong voltage source (like *Vdd, Ground, input or a highly capacitive bus*) to an output or a gate. Typically a stage consists of a logic gate along with all the pass transistors following it. Refer to Figure 1 for an example of a stage. Note that a chain of transistor channels forms an electrical path from a voltage source to the output of a stage. Thus, the flow of charge through a stage is conserved. One transistor in a stage is distinguished as the *trigger*. All the other transistors in a stage are assumed to be fully *on*, unless otherwise specified by the user. This gives rise to a worst case estimate.

In CMOS there are two components to the switching current, load current and short-circuit direct current. We deal with the direct current component in the next section. The peak load current through a stage is the minimum of capacities of all the transistors in the stage. The average charge flow through a stage is the product of total load capacitance and voltage swing. The average load current through a stage is estimated by dividing the average charge by the delay through that stage. The side paths from a stage are assumed to be *off*. In practice, in a structure like pass transistor array, not many such paths seem to be active at the same time.

In a typical run of *hercules*, the user indicates a clock signal transition (rise or fall) and wants to know the current requirements of this transition. Hercules finds all the nodes that can be affected

by this clock transition. These nodes are the basis of the initial set of stages. Each of these stages is handled in turn. For each stage, the average and peak load and direct current calculations are performed. Then all the stages this stage can activate, are recursively analyzed. At the end of this calculation, each node is tagged with its average & peak load current levels and average & peak direct current levels. The power is given by the product of *Vdd*, the charge drawn from the power supply and the frequency of this clock signal. Next, we present the algorithm.

procedure {POWERANALYSIS(*clock*)}
  while there are unvisited nodes in the *gate* adjacency list of *clock*
  do
    1. visit one of the gates A from the adjacency list.
      1.1 make the transistor T, gated by A, the *trigger* for a stage, $stage_A$.
      1.2 In a depth-first manner, expand $stage_A$, until a complete stage has been found.
      1.3 Perform delay and power analysis on this stage.
      1.4 Let n_i be the output node of $stage_A$. Recursively call POWERANALYSIS(n_i).
    2. Update the charge and current information for the node *clock*. Mark A as a visited gate.
  endwhile
endprocedure

In the Step 1.3 of this algorithm, the current levels from the children stages are combined with the current levels of a stage. For a node A which is either a bus or an input, the current requirements of all the stages with the node A are added to give the current flow at the node A. In other cases, the maximum current of all the stages a node participates in, is the current flow through that node. A node is considered to be a bus if its capacitance is above a threshold, or if user tags it to be a bus.

4 Short-Circuit Current Models

In CMOS static logic gates, there is a short-circuit current component when the input signal switches. As Veendrick [Vee84] observes, the average level of this current could be as large as the average level of the load current. Thus it is important to consider the short-circuit current component in a power analysis of a CMOS circuit.

Clearly the RC model would not be sufficient to model this component, because it assumes step function input signals. The rise time of the input signal, the gain, β, of the device and the load determine the total amount of charge flow through the short-circuit component. The key to an efficient implementation of this model is that all these factors can be combined into a number called *rise time ratio*, which equals the input signal rise time divided by the inherent output signal rise time. Inherent output signal rise time equals the rise time of the output node when a step

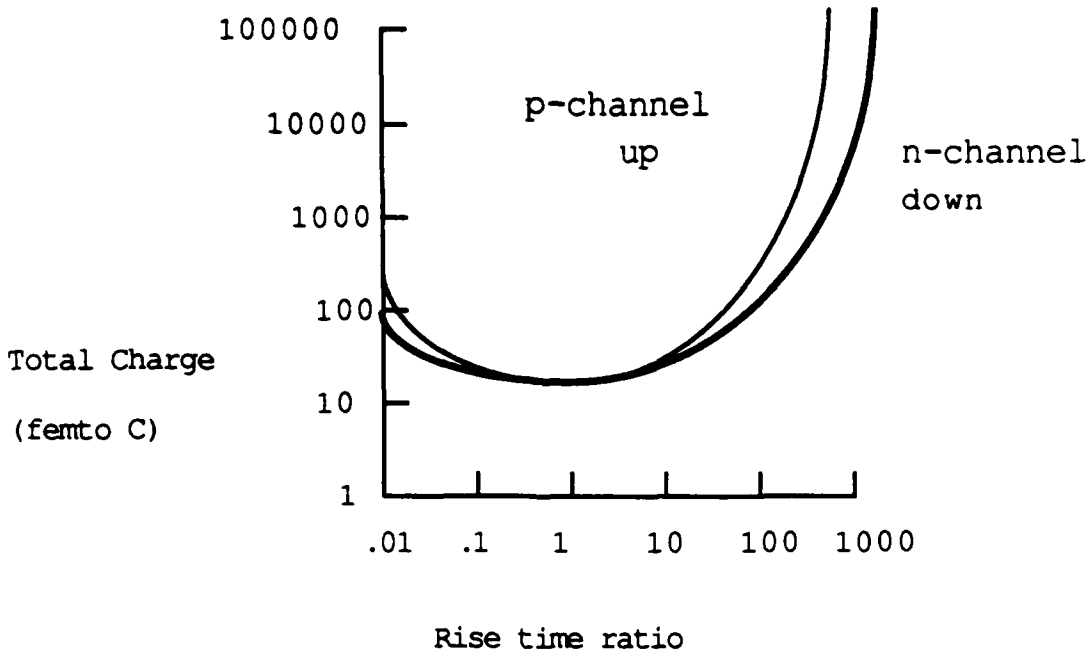


Figure 2: A Plot of Total Direct Charge Flow Vs. Rise Time Ratio

input is applied. The information about short-circuit current can be extracted from SPICE output on a device with different rise time ratio values. This can be stored in a table indexed by the rise time ratio. Hercules uses this table to estimate the current flow through a stage. All the transistors in a stage except the trigger are assumed to be fully turned on. The rise time ratio table is used only for estimating the direct current through the trigger. This technique works for the same reason as in Crystal. Most of the circuits are designed with few combinations of transistor and load sizes. SPICE runs on small pieces of circuits give almost all the information about the operating environment of these small circuits within a very large circuit.

Figure 2 shows the dependence of total charge flow on the rise time ratio for n-channel and p-channel devices. Note that the charge flow is directly proportional to the power consumption due to the short-circuit current. For the rise time ratio equalling 1, the total charge flow is minimized. Veendrick also makes the same observation in [Vee84] and uses this fact to design buffers optimal with respect to power consumption. The peak and average short-circuit currents are maximum when there is no load. Both of them tend to decrease with increasing load. The reason for that is that a part of short circuit current goes into driving the load. With a higher load, the amount of excess charge available to flow from *power* to *ground* is less. With rise time ratio equal to one, both the input and output signal speeds are equal. If the input is driven at a slower rate than this, then the extra time for which the inverter is in short-circuit mode ($V_{inv} - V_{th_n} \leq V_{in} \leq V_{inv} + V_{th_p}$) increases. This gives rise to higher direct charge flow as in Figure 2. On the other hand, if the input signal is faster than this, then the load can not absorb all of the short-circuit current. Thus, there is more of excess charge avialable to flow between the *power* and *ground*. In any case, Veendrick remarks that for a rise time ratio of one, the short-circuit dissipation would be less than 20% of

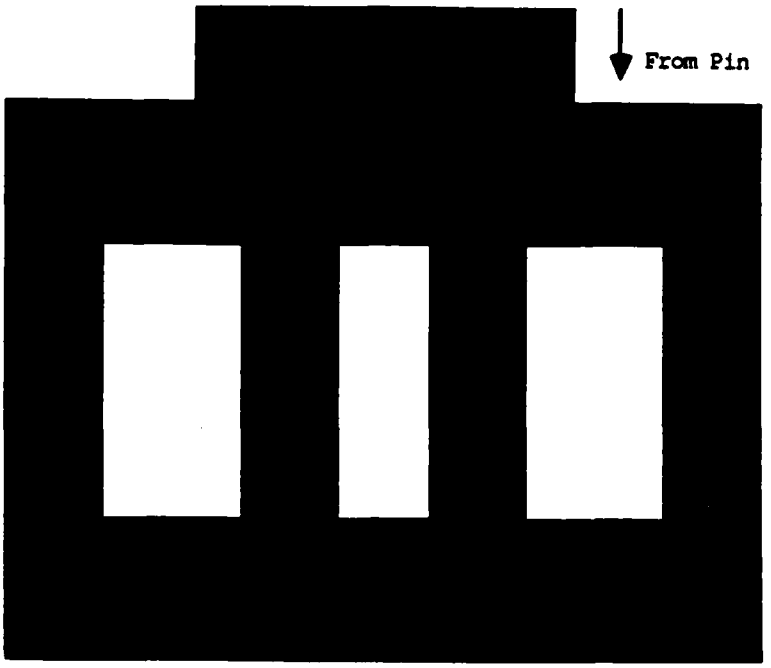


Figure 3: An Example of a *Comb* Structure

the total dissipation. Now we have a very good mathematical measure of what it means to have a good driving ratio between the driver size and the load. We can easily identify all the inverters (stages) with a rise time ratio within δ of 1, for a user specified δ. The user can choose a δ based on how finely tuned a system he wants.

5 Metal Bus Data Structure

Distributing *power* and *ground* with least resistance is one of the major parts of a design. Many a time, a circuit fails to perform because the drain terminals were receiving a voltage level significantly lower than *Vdd* and the source terminals were way above *ground*, although the external power supply was functioning properly.

The circuit extractor *mextra* [MJW83] had to be modified not to merge all the electrically equivalent metal rectangles into a single *Vdd* or *ground* node. It retains the information about the contact cuts from a metal bus to eitherr poly or diffusion layer. The information about the location of metal2 vias is also kept. For further information, the reader is referred to Wilson's thesis [Wil85].

Using the information retained by the mask geometry extractor, we construct a tree like data structure to model the metal rectangles connecting a *Vdd* pin to the drain terminals. Each node in the tree corresponds to a metal rectangle in the bus. It stores information about the resistivity, total current flow through it, contact cuts incident on it and its neighboring rectangles. The root of a tree points to the metal rectangle at the pin. A new node is created whenever the metal rectangle width changes, or an orthogonal rectangle intersects, or the metal layer changes. If *node*1 is the parent of *node*2 then the *node*2 metal rectangle connects with the *node*1 rectangle, and current

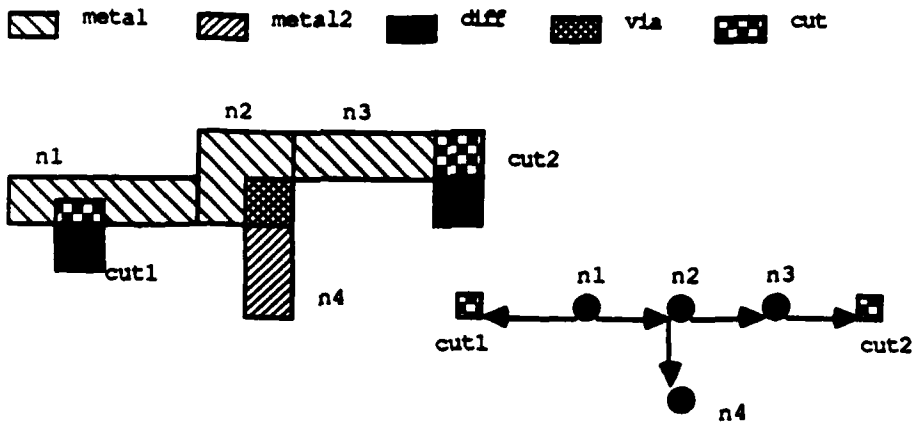


Figure 4: A Metal Tree Example

flows from (into) *node*1 into (from) *node*2 for a *Vdd* (*ground*) metal tree. Figure 4 illustrates this construction. The leaves of this tree correspond to the transistor terminals, *i.e.*, they contain only the contact cuts. Note that this tree is constructed in depth-first order connectivity of the metal bus. If it were a tree, then the voltage drop algorithm would be a very simple one. The power analysis program calculates current levels and tags each source (drain) terminal with its current requirements. The next step sums these values up the tree. Each node sums up the current requirements of its children, its own contact cuts and transmits this information to its parent. The root ends up with the total current requirements. In the next pass, we traverse down the tree calculating the voltage drop across the metal rectangle for that node. Note that both the current and the node resistance are known at this stage. At the end of this traversal, each contact cut ends up with the voltage drop from the pin to that node. The tree construction algorithm embeds many instances of the rectangle intersection problem, which involves reporting all the rectangles from a set of rectangles that intersect a given rectangle. We use a two dimension extension (by Wilson [Wil85]) of a group tree, originally proposed by McCreight [McC80], [McC81]. For a discussion on rectangle intersection algorithms the reader is referred to Ullman [Ull84] and Preparata, Shamos [PS85].

How should the loops in *power/gnd* structure be handled? If they are broken arbitrarily, it could lead to a large error. We observed that it is very seldom that a designer routes power with looping structures. However the pattern shown in Figure 3 is used quite often to feed power to an array of cells. The bottom line introducing the loops reinforces the distribution. Recall that a depth-first search algorithm can be easily modified to identify all the biconnected components in

a graph [AHU74] (A set of vertices forms a biconnected component if between any pair of vertices there are at least two edge disjoint paths). Each loop introduces one biconnected component. Thus, recognizing biconnenected components is equivalent to recognizing loops. The main objective of the whole excercise is to be able to distribute incoming current correctly along the outgoing edges at a node. We could do a complete Kirchoff Voltage Law (KVL) or Kirchoff Current Law (KCL) analysis of the whole metal network. But a bus network can have thousands of nodes in it. KVL or KCL analysis requires approximately n^3 time. On the other hand, a biconnected component has a very small fraction of nodes (typically an array could be of size 32, giving rise to approximately 96 node biconnected component). Doing the KVL analysis on a small number of such biconnected components seems to be a much more efficient solution.

We can really consider the solution techniques within biconnected components and the skeleton tree completely independently of each other. Thus we could use a simple heuristics within each component, if we have some information about the circuit. We found that even if we assign the outgoing currents only on the basis of conductances of the outgoing metal rectangles (rather than the whole path), we get a very good approximation for the comb structure in Figure 3. It is partly beacause of the uniformity in the sizes of each metal segment in such a structure.

Another issue we had to face concerns the frequency of the metal tree evaluation. When a stage is activated, it sources/sinks currents at a certain time instant. Ideally, the metal tree should be evaluated for each such time. But that does not gain us that much more information, specially given the cost of each evaluation. We average each current value over the number of cascaded stages in order to avoid overly pessimistic results.

6 Performance

Both the short-circuit current and load current models were validated with presently operational version of *Hercules*. Hercules was run with a control PLA and a 32 register file for a 32-bit microprocessor, QuarterHorse [HJK*85], designed at the University of Washington. The control PLA has 19 inputs, 68 outputs and 93 implicants. The register file has 32, 32-bit registers designed in the cross-coupled static style. Hercules was also tried on some 3 stage NAND and NOR networks. The results are encouraging. For all the cases, the average load current and the average direct current reported are within 25% of the SPICE reported numbers. However the peak load current was overestimated by as much as 100%. Recall that the peak load current is minimum of the peak currents of all the transistors in a stage. The peak direct current is calculated by dividing the total charge flow by the time input signal takes to go from $V_{inv} - V_{th_n}$ to $V_{inv} + V_{th_p}$. This number was also found to be off by as much as 80% on the higher side. We believe that we can do better than this with the peak load current calculation. We still do not know how to better predict the peak direct current.

7 nMOS Case

The Berkeley program *powest* assumes each device with a pullup load to be *on*. For every type of load device like depletion, the average fraction of time the device consumes direct current, $0 \leq f \leq 1$, is fixed in advance based on some experiments. For each such device, its direct current is multiplied by this fraction. Wilson's *pwranal* is more sophisticated in estimating DC current by identifying the chains of inverters or superbuffers. Only half the inverters in such a chain could be on and hence consuming direct current at any time. It also allows the user to initialize nodes. It propagates the initialized values as far in the circuit as it can. This can also reduce the power estimate. The *stage* based decomposition of a circuit is better suited to identify the number of pullup devices that could be on simultaneously. Note that an inverter and the pass transistors following it are lumped into a single stage as in Figure 1. Thus in a chain of stages only half the stages could be on. This allows for the detection of alternation in more general structures than inverters.

8 Acknowledgements

The idea that max-flow could be useful in current analysis was suggested to me by my advisor, Larry Snyder. I would like to thank him for many invaluable discussions. The need to consider the direct current component in CMOS was pointed out to me by Bill Beckett. This work has benefitted a lot from discussions with Bill Beckett and Larry McMurchie. The presentation of this paper has improved considerablly due to Larry McMurchie's comments. I also wish to acknowledge the influence of John Ousterhout's and Jeffrey Wilson's work on the way *Hercules* evolved.

References

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.

[Cme82] R. Cmelik. *Berkeley VLSI CAD Tools Manual.* Computer Science Dept., University of California, Berkeley, 1982.

[HJK*85] S. Ho, B. Jinks, T. Knight, J. Schaad, L. Snyder, A. Tyagi, and C. Yang. The Quarter Horse: A Case Study in Rapid Prototyping of a 32-bit Microprocessor Chip. In *IEEE Proceedings of the International Conference on Computer Design: VLSI in Computer*, IEEE Computer Society, 1985.

[Jou83] N.P. Jouppi. Timing Analysis for nMOS VLSI. In *Proceedings of 20th Design Automation Conference*, ACM-IEEE, 1983.

[McC80] E.M. McCreight. *Efficient Algorithms for Enumerating Intersecting Rectangles and Intervals.* Technical Report CSL-80-9, Xerox PARC, Palo Alto, 1980.

[McC81] E.M. McCreight. *Priority Search Trees.* Technical Report CSL-81-5, Xerox PARC, Palo Alto, 1981.

[MJW83] R.N. Mayo, Ousterhout J.K., and Scott W.S. *1983 VLSI Tools Manual.* Report No. UCB/CSD83/115, Computer Science Dept., University of California, Berkeley, 1983.

[Ous83] J.K. Ousterhout. Crystal: A Timing Analyzer for nMOS VLSI Circuits. In *Proceedings of 3rd Caltech Conference on VLSI*, Computer Science Press, 1983.

[Ous84] J.K. Ousterhout. Switch-Level Delay Models for Digital MOS VLSI. In *Proceedings of 21st Design Automation Conference*, ACM-IEEE, 1984.

[Ous85] J.K. Ousterhout. A Switch-Level Timing Verifier for Digital MOS VLSI. *IEEE Transactions on Computer Aided Design*, July 1985.

[PS85] F.P. Preparata and M.I. Shamos. *Computational Geometry, an Introduction.* Springer-Verlag, New York, NY, 1985.

[ST86] L. Snyder and A. Tyagi. The Energy Complexity of Transitive Functions. In *Proceedings of 24th Allerton Conference on Communication, Control and Computing*, Allerton, Illinois, 1986.

[Ull84] J.D. Ullman. *Computational Aspects of VLSI.* Computer Science Press, Rockville, Md., 1984.

[Vee84] H. J. M. Veendrick. Short-Circuit Dissipation of Static CMOS Circuitry and Its Impact on the Design of Buffer Circuits. *IEEE Journal of Solid State Circuits*, August 1984.

[Wil85] J. Wilson. *Analysis of Power Requirements inside of nMOS Integrated Circuits.* M.S. Thesis, Computer Science Dept., Oregon Graduate Center, Beaverton, 1985.